Programmieren mit Python

Spiele, Grafik, Daten und Sensoren mit PBC+ Studio



AB 14 JAHREN

DIE PROGRAMMIERREIHE FÜR DEN PICOBOY COLOR PLUS

Dieses Buch wurde mit künstlicher Intelligenz (KI) generiert. Die Programme wurden getestet – Fehler sind trotzdem möglich.

Bevor es losgeht

In diesem Buch lernst du, in **Python** zu programmieren – einer der meistgenutzten Programmiersprachen der Welt. Python steckt in Webseiten, wissenschaftlichen Auswertungen, Robotern und künstlicher Intelligenz. Du lernst die Sprache an einem kleinen Gerät, das du in der Hand halten kannst: dem **PicoBoy Color Plus** mit Farbbildschirm, Tasten, LEDs, Lautsprecher und Bewegungssensor. So siehst du sofort, was dein Programm tut.

Du brauchst keine Vorkenntnisse. Am Anfang schreibst du Texte auf den Bildschirm, dann rechnest du, zeichnest, reagierst auf Tasten, speicherst Daten und lässt Figuren über den Bildschirm laufen. Am Ende baust du ein eigenes Spiel oder Programm.

So ist das Buch aufgebaut

Jedes Kapitel hat denselben Aufbau:

- **Worum geht's?** – was du in dem Kapitel lernst.
- **Erklärungen** mit kurzen Programmen. Jedes Programm hat Zeilennummern, und der Text erklärt, was in welcher Zeile passiert. Daneben siehst du, was im Emulator erscheint, oder darunter, was im Terminal steht.
- **Ausprobieren** – ein etwas größeres Programm, das das Kapitel zusammenfasst.
- **Übungsaufgaben** mit Sternen: ⭐ zum Einsteigen, ⭐⭐ wenn du weiter willst, ⭐⭐⭐ zum Knobeln. Die Lösungen stehen in Anhang C. Schau erst nach, wenn du es selbst versucht hast!
- **Das Wichtigste in Kürze** – zum Wiederholen.

Unterwegs findest du farbige Kästen:

> **Tipp:** Hier steht, wie etwas leichter geht.

> **Achtung:** Hier steht, worauf du aufpassen musst – oft mit der Fehlermeldung, die dann erscheint.

> **Info:** Hier steht etwas, das gut zu wissen ist.

Was du brauchst

- einen Computer mit dem Browser **Chrome** oder **Edge**
- die Seite **pbcstudio.de** (PBC+ Studio ab Build 110)
- einen **PicoBoy Color Plus** mit USB-Kabel – fast alles geht aber auch ohne Gerät im **Emulator**, einem PicoBoy auf dem Bildschirm

Alle Programme aus dem Buch gibt es als `.py`-Dateien. Den Dateinamen findest du über jedem Programm. Abtippen lohnt sich trotzdem: Beim Tippen lernst du die Sprache am schnellsten – und lernst nebenbei, Tippfehler zu finden.

Die Teile des Buchs

Teil	Darum geht es
I – Los geht's	Gerät und PBC+ Studio, das erste Programm, Ein- und Ausgabe in der Konsole
II – Grundlagen	Variablen, Zeichnen, Schleifen, Entscheidungen, Tasten, LEDs und Töne, Texte
III – Größere Programme	Funktionen, Rückgabewerte, Listen, Dateien
IV – Grafik und Bewegung	Turtle, Canvas, Sprites, Kollisionen, Bewegungssensor
V – Projekte	ein Spiel Schritt für Schritt – und dann dein eigenes Projekt
Zusatzkapitel	Objekte und Klassen

Viel Spaß beim Programmieren!

Inhalt

Teil I – Los geht's

1 Der PicoBoy Color Plus und PBC+ Studio 5
2 Das erste Programm 11
3 Die Konsole: `print` und `input` 20

Teil II – Grundlagen

4 Variablen und Rechnen 29
5 Zeichnen und Farben 38
6 Schleifen und Zeit 47
7 Entscheidungen 57
8 Tasten und Joystick 68
9 LEDs und Töne 79
10 Texte (Strings) 89

Teil III – Größere Programme bauen

11 Funktionen 98
12 Funktionen mit Rückgabewert 107
13 Listen 118
14 Listen im Spiel 128
15 Dateien: Daten dauerhaft speichern 138

Teil IV – Grafik und Bewegung

16 Turtle-Grafik 149
17 Der Canvas – flimmerfrei zeichnen 159
18 Sprites – Figuren, die sich bewegen 168
19 Kollisionen 177
20 Der Bewegungssensor 185

Teil V – Projekte

21 Projekt: Ein Spiel Schritt für Schritt 193
22 Dein eigenes Projekt 203

Zusatzkapitel

Z Objekte und Klassen 213

Anhang

A Befehlsübersicht 227
B Fehlermeldungen verstehen 235
C Lösungen der Übungsaufgaben 247
D Für Fortgeschrittene: Bilder, Dateien und Firmware 322

KAPITEL 1

Der PicoBoy Color Plus und PBC+ Studio

Worum geht's? Bevor du programmierst, lernst du dein Werkzeug kennen: den PicoBoy Color Plus, einen kleinen Computer mit Farbbildschirm und Tasten, und PBC+ Studio, die Programmierumgebung im Browser. Am Ende des Kapitels hast du ein erstes Python-Programm gestartet, gespeichert und wieder geöffnet.

1.1 Das Gerät

Der **PicoBoy Color Plus** (kurz: PBC+) ist ein kleiner Computer, den man in der Programmiersprache **Python** programmiert. Er sieht aus wie eine Spielkonsole, und du kannst darauf auch eigene Spiele schreiben. Er kann aber noch mehr: messen, rechnen, Töne spielen, Daten speichern.



Teil	Wofür?
Bildschirm	Farbbildschirm, 240 Pixel breit und 280 Pixel hoch
Joystick	kleiner Steuerknüppel für hoch, runter, links und rechts. Drückst du ihn nach unten, ist das die Taste **MITTE**.
Taste A und Taste B	zwei Knöpfe, z. B. zum Springen oder Bestätigen
LEDs rot, gelb, grün	drei kleine Lämpchen neben dem Bildschirm
RGB-LED	ein Lämpchen, das in jeder Farbe leuchten kann

Teil	Wofür?
Lautsprecher	spielt Töne und Melodien
Bewegungssensor	misst, wie das Gerät gehalten, gekippt oder geschüttelt wird
I²C-Buchse	weiße Buchse, passend für STEMMA QT und Qwiic: Sensoren und andere I²C-Module anstecken, ohne Löten

Mit dem Computer verbindest du den PBC+ über ein **USB-Kabel**. Auf dem Gerät läuft **MicroPython**, eine Variante von Python für kleine Computer. Fast alles, was du in diesem Buch über Python lernst, gilt genauso für Python auf einem PC.

1.2 PBC+ Studio im Browser öffnen

Programmiert wird im Browser mit **PBC+ Studio**. Öffne dafür die Seite **pbcstudio.de**, am besten in **Chrome** oder **Edge**. Nur diese Browser können sich über USB mit dem Gerät verbinden. Du musst nichts installieren.

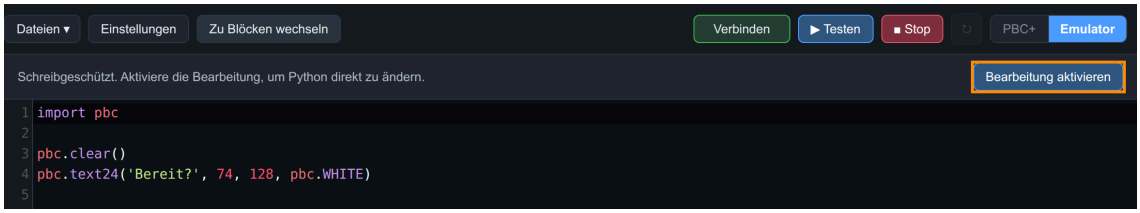
PBC+ Studio kann Programme auf zwei Arten darstellen: als Blöcke zum Zusammenstecken oder als Text. In diesem Buch schreibst du Python als **Text**. Beim allerersten Öffnen zeigt PBC+ Studio aber die Blockansicht mit einem kleinen Startprogramm. Du wechselst deshalb einmal in den Textmodus.

1.3 In den Textmodus wechseln – der Python-Editor

1. Klicke oben auf **Zu Text wechseln**.



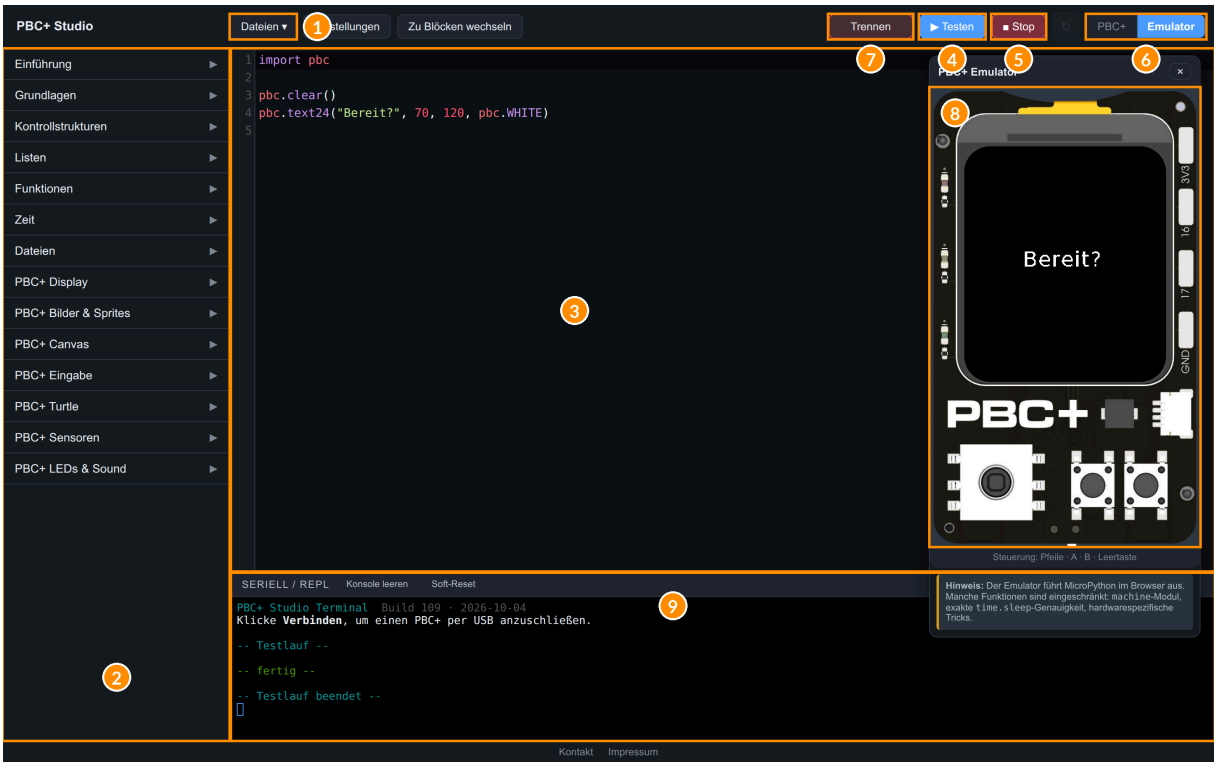
2. Du siehst jetzt das Startprogramm als Python-Text. Es ist zunächst schreibgeschützt. Klicke rechts auf **Bearbeitung aktivieren** und bestätige die Rückfrage mit **OK**.



3. Fertig: Du kannst jetzt im Editor schreiben. PBC+ Studio merkt sich den Textmodus. Wenn du die Seite später wieder öffnest, bist du gleich im Textmodus und dein letztes Programm ist noch da.

> Der Knopf heißt im Textmodus **Zu Blöcken wechseln**. Klicke ihn nicht an: Beim Zurückwechseln gehen alle Änderungen an deinem Python-Text verloren. PBC+ Studio fragt vorher nach, dann wählst du **Abbrechen**.

So sieht PBC+ Studio im Textmodus aus:



Nr.	Name	Wofür?
1	**Dateien**	Programme speichern und öffnen, Bilder hochladen
2	**Hilfe-Leiste**	Erklärungen und Beispiele zu Python und zu allen PBC±Befehlen, sortiert nach Themen
3	**Editor**	Hier schreibst du dein Programm. Links stehen die Zeilennummern.
4	**▶ Testen**	startet dein Programm
5	**■ Stop**	hält dein Programm an
6	**Ziel**	Läuft das Programm im **Emulator** oder auf dem echten Gerät (**PBC+**)?
7	**Verbinden / Trennen**	verbindet PBC+ Studio mit dem Gerät oder dem Emulator
8	**Emulator**	ein PicoBoy auf dem Bildschirm, der so tut, als wäre er das echte Gerät
9	**Terminal**	Hier erscheinen Ausgaben deines Programms und Fehlermeldungen. Man nennt es auch **Konsole**.

Die Hilfe-Leiste: Klickst du links auf ein Thema, z. B. **PBC+ Display**, klappt eine Liste mit Befehlen auf. Ein Klick auf einen Befehl öffnet rechts eine Erklärung mit Beispiel. Mit **In den Editor einfügen** landet das Beispiel an der Stelle im Editor, an der der Cursor steht.



1.4 Testen im Emulator oder auf dem Gerät

Lösche alles im Editor (**Strg + A**, dann **Entf**) und tippe dieses kleine Programm ab:

k01_bereit.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Bereit?", 70, 120, pbc.WHITE)
```



So sieht es im Emulator aus.

Was die Zeilen bedeuten, lernst du in Kapitel 2. Achte beim Abtippen auf jedes Zeichen: Klammern, Kommas, Anführungszeichen und Groß- und Kleinschreibung.

Im Emulator testen:

1. Wähle oben rechts als Ziel **Emulator**. Das Emulator-Fenster öffnet sich.
2. Klicke auf ▶ **Testen**. Auf dem Bildschirm des Emulators erscheint „Bereit?“. Im Terminal steht `-- Testlauf --` und am Ende `-- fertig --`.
3. Mit ■ **Stop** hältst du ein Programm an, das nicht von selbst aufhört.

Das Emulator-Fenster kannst du an seiner Titelleiste verschieben. Die Tasten des Emulators drückst du mit der Tastatur oder mit der Maus:

Gerät	Tastatur
Joystick hoch, runter, links, rechts	Pfeiltasten
Joystick drücken (MITTE)	Leertaste
Taste A / Taste B	Taste **A** / Taste **B**

Auf dem Gerät testen:

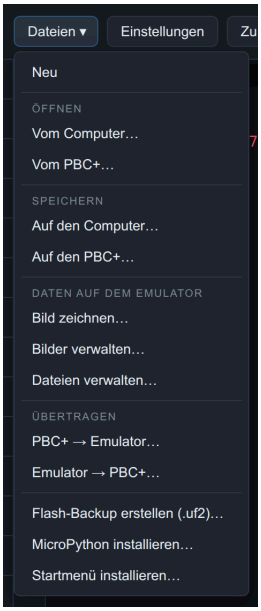
1. Verbinde den PBC+ mit dem USB-Kabel mit dem Computer und schalte ihn ein.
2. Wähle als Ziel **PBC+** und klicke auf **Verbinden**. Der Browser zeigt eine Liste, wähle dort den PicoBoy aus.
3. Klicke auf ▶ **Testen**. Dein Programm läuft jetzt auf dem echten Gerät.

> **Emulator oder Gerät?** Der Emulator ist praktisch, weil du kein Kabel brauchst. Er ist ungefähr so schnell wie das echte Gerät. Manches geht aber nur auf dem Gerät: Kippen und Schütteln (im Emulator liegt das Gerät immer flach auf dem Tisch), echte Töne aus dem Lautsprecher und Dateien, die auch nach dem Ausschalten erhalten bleiben.

1.5 Programme als `.py` speichern und öffnen

PBC+ Studio merkt sich dein Programm, auch wenn du die Seite neu lädst. Das gilt aber nur in diesem Browser auf diesem Computer. Damit nichts verloren geht, speicherst du dein Programm als Datei:

- **Speichern: Dateien → Auf den Computer…** Das Programm landet als Datei `programm.py` im Download-Ordner. Gib ihr am besten gleich einen passenden Namen, zum Beispiel `bereit.py`. Die Endung `.py` steht für Python.
- **Öffnen: Dateien → Vom Computer…** und die Datei auswählen. Der Inhalt ersetzt das Programm im Editor.
- **Neu anfangen: Dateien → Neu** leert den Editor. Vorher speichern!



Eine `.py`-Datei ist eine ganz normale Textdatei. Du kannst sie auch mit jedem anderen Texteditor öffnen oder mit einem Python auf dem PC ausführen, dort fehlen dann allerdings die PBC±Befehle.

1.6 Ein Programm auf dem Gerät behalten

Mit ▶ **Testen** läuft ein Programm nur so lange, bis du den PBC+ ausschaltest oder ein anderes Programm testest. Soll es auf dem Gerät bleiben, speicherst du es dort:

1. Verbinde das Gerät (Ziel **PBC+**, **Verbinden**).
2. **Dateien → Auf den PBC+…**, gib einen Dateinamen ein, z. B. `bereit.py`, und klicke auf **Speichern**.
3. Einmalig: **Dateien → Startmenü installieren…**

Danach zeigt der PBC+ beim Einschalten ein Menü mit allen gespeicherten Programmen. Mit dem Joystick wählst du eins aus und startest es. Mit **Dateien → Vom PBC+...** holst du ein Programm vom Gerät zurück in den Editor.

Übungsaufgaben

> **Aufgabe 1** ⭐ **Teile finden** Welche Taste der Tastatur ist im Emulator der Joystick-Druck (MITTE)? Mit welchem Knopf hältst du ein Programm an? Wo erscheinen Fehlermeldungen?

> **Aufgabe 2** ⭐ **Dein Name** Ändere das Programm aus 1.4: Statt `Bereit?` soll `Hallo,` und dein Name erscheinen, und statt `pbc.WHITE` schreibst du `pbc.YELLOW`. Passt der Text nicht auf den Bildschirm, mach die erste Zahl (`70`) kleiner.

> **Aufgabe 3** ⭐ **Speichern und wieder öffnen** Speichere dein Programm aus Aufgabe 2 als `hallo.py`. Klicke auf **Dateien → Neu** und öffne danach deine Datei wieder. Ist alles noch da?

> **Aufgabe 4** ⭐⭐ **Die Hilfe benutzen** Öffne in der Hilfe-Leiste **PBC+ Display** und dort den Eintrag mit `text12 / 18 / 24`. Füge das Beispiel mit **In den Editor einfügen** unter dein Programm ein und teste. Welche der drei Schriften ist die größte?

> **Aufgabe 5** ⭐⭐⭐ **Auf das echte Gerät** *(wenn du einen PicoBoy hast)* Teste dein Programm auf dem Gerät. Speichere es danach mit **Dateien → Auf den PBC+...** auf dem Gerät, schalte den PicoBoy aus und wieder ein und starte dein Programm aus dem Startmenü.

Die Lösungen stehen in Anhang C.

> **Das Wichtigste in Kürze**
>
> - Der PicoBoy Color Plus wird in **Python** (genauer: MicroPython) programmiert.
> - PBC+ Studio läuft im Browser unter **pbcstudio.de** (Chrome oder Edge).
> - Einmalig **Zu Text wechseln** und **Bearbeitung aktivieren**, danach schreibst du im Editor.
> - ▶ **Testen** startet, ■ **Stop** hält an. Ziel **Emulator** oder **PBC+**.
> - Ausgaben und Fehlermeldungen stehen im **Terminal**.
> - Programme speicherst du als `.py`-Datei: **Dateien → Auf den Computer...** bzw. **Auf den PBC+...**

KAPITEL 2

Das erste Programm

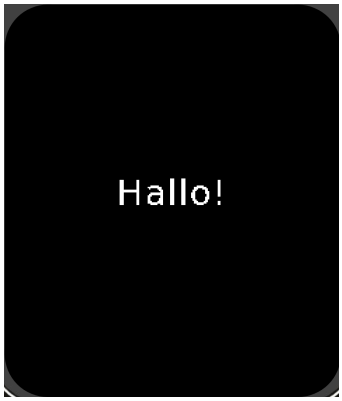
Worum geht's? Du schreibst Texte in verschiedenen Größen und Farben auf den Bildschirm und lernst, wie der PicoBoy Positionen auf dem Bildschirm angibt. Außerdem lernst du, Fehlermeldungen zu lesen. Das wirst du beim Programmieren ständig brauchen.

2.1 `import pbc` – Befehle für den PicoBoy holen

Ein Python-Programm ist eine Folge von **Anweisungen**. Der Computer führt sie von oben nach unten aus, eine nach der anderen. Hier ist ein vollständiges Programm:

k02_hallo.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Hallo!", 80, 120, pbc.WHITE)
```



Die erste Zeile `import pbc` lädt das **Modul** `pbc`. Ein Modul ist eine Sammlung fertiger Befehle. Python selbst weiß nichts über den Bildschirm oder die Tasten des PicoBoy. Diese Befehle stecken im Modul `pbc`. Nach dem Import erreichst du sie mit `pbc.` und dem Namen des Befehls, zum Beispiel `pbc.clear()`.

Ohne `import pbc` kennt Python den Namen `pbc` nicht. Das Programm bricht in der ersten Zeile ab, die `pbc` benutzt:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pbc' isn't defined
```

Die Zeile `import pbc` gehört deshalb in jedes Programm in diesem Buch, und zwar ganz nach oben.

Die leere Zeile 2 hat keine Bedeutung für Python. Sie trennt nur den Import vom Rest und macht das Programm übersichtlicher.

2.2 Bildschirm löschen und Text schreiben

`pbc.clear()` löscht den Bildschirm: Er wird schwarz. Die runden Klammern zeigen, dass hier ein Befehl **aufgerufen** wird. Solche Befehle heißen in Python **Funktionen**. Auch wenn nichts in den Klammern steht, müssen sie da sein.

`pbc.text24("Hallo!", 80, 120, pbc.WHITE)` schreibt einen Text. In den Klammern stehen vier Angaben, durch Kommas getrennt. Man nennt sie **Argumente**:

Argument	Bedeutung
`"Hallo!"`	der Text, in Anführungszeichen
`80`	wie weit von links (x)
`120`	wie weit von oben (y)
`pbc.WHITE`	die Farbe

Die Reihenfolge der Argumente ist fest. Vertauschst du sie, versteht der Befehl etwas anderes oder meldet einen Fehler.

Texte stehen in Python in Anführungszeichen. Python nennt sie **Strings**. In diesem Buch benutzen wir gerade Anführungszeichen `"…"`, wie sie die Tastatur mit **Umschalt + 2** schreibt.

Farben: Das Modul `pbc` kennt elf fertige Farben: `pbc.BLACK`, `pbc.WHITE`, `pbc.RED`, `pbc.GREEN`, `pbc.BLUE`, `pbc.YELLOW`, `pbc.CYAN`, `pbc.MAGENTA`, `pbc.ORANGE`, `pbc.PURPLE` und `pbc.GRAY`. Gibst du `pbc.clear()` eine Farbe mit, wird der Bildschirm in dieser Farbe gefüllt:

k02_farben.py

```
1 import pbc
2
3 pbc.clear(pbc.BLUE)
4 pbc.text18("Gelb auf Blau", 20, 40, pbc.YELLOW)
5 pbc.text18("Weiss", 20, 80, pbc.WHITE)
6 pbc.text18("Orange", 20, 120, pbc.ORANGE)
```

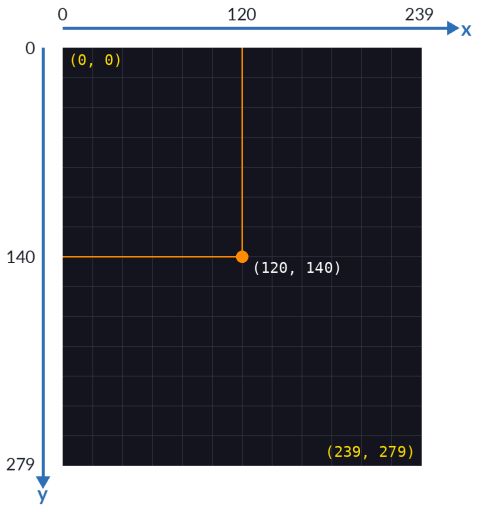


> **Keine Umlaute auf dem Bildschirm.** Die Schrift des PicoBoy kennt kein ä, ö, ü und ß. Statt eines Umlauts erscheinen zwei Fragezeichen. Schreibe deshalb „Weiss“, „Gruesse“, „schoen“.

2.3 Das Koordinatensystem (240 × 280 Pixel)

Der Bildschirm besteht aus kleinen Punkten, den **Pixeln**: 240 nebeneinander und 280 untereinander. Jedes Pixel hat eine Adresse aus zwei Zahlen, seine **Koordinaten** (x, y):

- **x** zählt von **links** nach rechts: 0 bis 239.
- **y** zählt von **oben** nach unten: 0 bis 279.
- (0, 0) ist die linke obere Ecke.



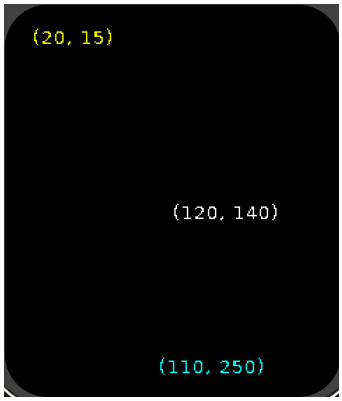
Anders als im Matheunterricht wächst y also nach **unten**. Das ist bei Bildschirmen üblich, weil man Text auch von oben nach unten liest.

Bei Texten geben x und y die **linke obere Ecke** des Textes an. Das folgende Programm schreibt die Koordinaten genau an die Stelle, die sie beschreiben:

k02_koordinaten.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text12("(20, 15)", 20, 15, pbc.YELLOW)
5 pbc.text12("(120, 140)", 120, 140, pbc.WHITE)
6 pbc.text12("(110, 250)", 110, 250, pbc.CYAN)
```

> Die Ecken des Bildschirms sind abgerundet. Was ganz in eine Ecke geschrieben wird, ist teilweise abgeschnitten. Halte deshalb etwa 10 bis 15 Pixel Abstand zum Rand.



2.4 Schriftgrößen: `text`, `text12`, `text18`, `text24`

Für Texte gibt es vier Funktionen:

k02_groessen.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text("text: 8 Pixel", 10, 20, pbc.WHITE)
5 pbc.text12("text12: 12 Pixel", 10, 50, pbc.WHITE)
6 pbc.text18("text18: 18 Pixel", 10, 80, pbc.WHITE)
7 pbc.text24("text24", 10, 120, pbc.WHITE)
```



- `pbc.text12`, `pbc.text18`, `pbc.text24`: eine gut lesbare Schrift, bei der jeder Buchstabe nur so breit ist, wie er sein muss (ein „i“ ist schmaler als ein „m“). Die Zahl ist ungefähr die Höhe einer Zeile in Pixeln.
- `pbc.text`: eine kleine, kantige Schrift, bei der **jedes** Zeichen genau 8 Pixel breit und 8 Pixel hoch ist. Sie eignet sich für Tabellen, weil die Zeichen genau untereinander stehen.

Wie breit ein Text ungefähr wird, siehst du am schnellsten durch Ausprobieren. Ein Text, der über den rechten Rand hinausgeht, wird einfach abgeschnitten.

2.5 Kommentare mit `#`

Alles, was hinter einem `#` steht, ignoriert Python. Solche Notizen heißen **Kommentare**. Sie erklären Menschen, was ein Programm tut:

k02_kommentare.py

```
1 # Begruessung fuer den Tag der offenen Tuer
2 import pbc
3
4 pbc.clear()                                    # Bildschirm schwarz
5 pbc.text24("Willkommen", 50, 100, pbc.YELLOW)
6 # pbc.text12("Raum 104", 85, 140, pbc.WHITE)     # noch nicht fertig
7 pbc.text12("in der Informatik", 70, 150, pbc.WHITE)
```

- Zeile 1 ist ein Kommentar über das ganze Programm.
- In Zeile 4 steht der Kommentar hinter einer Anweisung. Die Anweisung wird ausgeführt, der Kommentar nicht.
- Zeile 6 ist eine Anweisung, die mit `#` „auskommentiert“ wurde. Sie wird nicht ausgeführt, bleibt aber im Programm stehen. So kannst du etwas vorübergehend abschalten, ohne es zu löschen.



2.6 Wenn etwas schiefgeht: Fehlermeldungen lesen

Beim Programmieren macht jeder Fehler, auch Profis, jeden Tag. Wichtig ist, dass du sie schnell findest. Python hilft dir dabei mit einer **Fehlermeldung** im Terminal. Zusätzlich öffnet PBC+ Studio ein Hinweisfenster mit einer Erklärung in einfachen Worten.

Beispiel 1: ein Tippfehler im Namen

k02_fehler_tippfehler.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.txt24("Hallo!", 80, 120, pbc.WHITE)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
AttributeError: module 'pbc' has no attribute 'txt24'
```

So liest du die Meldung, am besten von **unten nach oben**:

1. Die letzte Zeile nennt die **Fehlerart** und eine Beschreibung: `AttributeError` heißt, dass es etwas mit diesem Namen nicht gibt. Das Modul `pbc` hat keinen Befehl `txt24`.
2. Darüber steht, **in welcher Zeile** das passiert ist: `line 4`. Die Zeilennummer findest du links im Editor.
3. `Traceback (most recent call last)` ist nur die Überschrift.

Beispiel 2: Groß- und Kleinschreibung

k02_fehler_name.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Hallo!", 80, 120, pbc.WHITE)
5 pbc.text12("Wie geht's?", 80, 160, pbc.white)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
AttributeError: module 'pbc' has no attribute 'white'
```

Python unterscheidet streng zwischen Groß- und Kleinbuchstaben. `pbc.WHITE` ist eine Farbe, `pbc.white` gibt es nicht. Beachte: Zeile 4 wurde noch ausgeführt, „Hallo!“ steht also auf dem Bildschirm. Das Programm bricht erst in der Zeile mit dem Fehler ab.

Beispiel 3: eine Klammer fehlt

k02_fehler_klammer.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Hallo!", 80, 120, pbc.WHITE
5 pbc.text12("Wie geht's?", 80, 160, pbc.WHITE)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 5
SyntaxError: invalid syntax
```

Ein `SyntaxError` bedeutet: Python kann das Programm nicht einmal lesen. Hier fehlt am Ende von Zeile 4 die schließende Klammer. Bei einem `SyntaxError` läuft **keine einzige** Zeile, der Bildschirm bleibt so, wie er war.

> Bei einem `SyntaxError` liegt der Fehler in der genannten Zeile **oder kurz davor**. Eine vergessene Klammer oder ein vergessenes Anführungszeichen merkt Python manchmal erst in der nächsten Zeile.

Die drei häufigsten Fehler am Anfang:

Meldung	Bedeutung	Typische Ursache
`SyntaxError`	Python versteht die Zeile nicht	fehlende Klammer, fehlendes Anführungszeichen, fehlendes Komma
`NameError`	einen Namen gibt es nicht	`import pbc` vergessen, Name falsch geschrieben
`AttributeError`	einen Befehl oder Wert *in einem Modul* gibt es nicht	`pbc.txt24`, `pbc.white`

Ausprobieren: Eine Visitenkarte

k02_visitenkarte.py

```
1  # Visitenkarte
2  import pbc
3
4  pbc.clear(pbc.WHITE)
5
6  # Name gross und farbig
7  pbc.text24("Mia Weber", 20, 30, pbc.BLUE)
8
9  # Angaben darunter
10 pbc.text12("Klasse 8b", 20, 80, pbc.BLACK)
11 pbc.text12("Lieblingsfach: Informatik", 20, 100, pbc.BLACK)
12 pbc.text12("Hobby: Klettern", 20, 120, pbc.BLACK)
13
14 # Fusszeile in kleiner Festbreitenschrift
15 pbc.text("PicoBoy Color Plus", 20, 250, pbc.GRAY)
```

- Zeile 4 füllt den Bildschirm weiß. Darauf ist schwarze und blaue Schrift gut zu lesen.
- Die Kommentare teilen das Programm in Abschnitte: Name, Angaben, Fußzeile.
- Die drei Angaben stehen jeweils 20 Pixel untereinander (y = 80, 100, 120). Bei `text12` ist das ein angenehmer Zeilenabstand.



Zum Verändern: Schreib deine eigene Visitenkarte. Probiere andere Hintergrundfarben. Welche Schriftfarben sind darauf gut lesbar?

Übungsaufgaben

Aufgabe 1 ⭐ **Wo erscheint was?** Lies das Programm, ohne es zu testen. Wo auf dem Bildschirm erscheinen die drei Wörter, und in welcher Farbe? Skizziere den Bildschirm und prüfe danach im Emulator.

k02_aufgabe1.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text18("oben", 20, 20, pbc.WHITE)
5 pbc.text18("Mitte", 95, 130, pbc.GREEN)
6 pbc.text18("unten", 150, 240, pbc.RED)
```

Aufgabe 2 ⭐ **Drei Zeilen über dich** Schreibe ein Programm, das deinen Vornamen groß (`text24`) und darunter deine Klasse und ein Hobby (`text18`) zeigt, alles in verschiedenen Farben.

Aufgabe 3 ⭐⭐ **Finde die Fehler** Dieses Programm enthält drei Fehler. Teste es, lies die Fehlermeldung, korrigiere den Fehler und teste erneut, bis alles läuft. Notiere zu jedem Fehler die Fehlerart.

k02_aufgabe3.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.Text24("Sommerfest", 30, 60, pbc.YELLOW)
5 pbc.text18("Freitag, 14 Uhr, 100, 120, pbc.WHITE)
6 pbc.text18("Schulhof", 100, 150, pbc.white)
```

Aufgabe 4 ⭐⭐ **Ein Schild** Schreibe ein Programm, das den Bildschirm rot füllt und in der Mitte groß „STOP“ in Weiß zeigt. Darunter steht klein „Bitte warten“. Probiere so lange verschiedene x-Werte aus, bis die Texte ungefähr in der Mitte stehen.

Aufgabe 5 ⭐⭐⭐ Stundenplan Zeige deinen Stundenplan für einen Tag als Tabelle mit den Spalten Stunde, Fach und Raum. Nimm dafür `pbc.text` und überlege: Wie viele Leerzeichen brauchst du, damit die Spalten genau untereinander stehen? Wie groß muss der Abstand der Zeilen mindestens sein?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `import pbc` steht ganz oben in jedem Programm. Danach erreichst du die PicoBoy-Befehle mit `pbc.…`.
- Anweisungen werden von oben nach unten ausgeführt.
- `pbc.clear()` löscht den Bildschirm, `pbc.clear(farbe)` füllt ihn.
- `pbc.text12/18/24(text, x, y, farbe)` und `pbc.text(text, x, y, farbe)` schreiben Text. Texte (Strings) stehen in `"…"`.
- Koordinaten: x von links (0–239), y von **oben** (0–279).
- `#` leitet einen Kommentar ein.
- Fehlermeldungen von unten lesen: Fehlerart, dann Zeilennummer.

KAPITEL 3

Die Konsole: `print` und `input`

Worum geht's? Bisher hat dein Programm nur auf den Bildschirm des PicoBoy geschrieben. Jetzt lernst du die Konsole kennen: Dort kann dein Programm Texte ausgeben, Fragen stellen und Antworten entgegennehmen. Du lernst, wie man Texte in Zahlen umwandelt und umgekehrt, und probierst Python direkt Zeile für Zeile aus.

3.1 Ausgaben in der Konsole mit `print`

Unten in PBC+ Studio liegt das **Terminal**, auch **Konsole** genannt. Dort hast du schon Fehlermeldungen gesehen. Mit der Funktion `print` kann dein Programm selbst etwas in die Konsole schreiben:

k03_print.py

```
1 # Ein Wetterbericht in der Konsole
2 print("Wetterbericht für Freitag")
3 print("------------------------")
4 print("Morgens:", "Nebel,", 8, "Grad")
5 print("Mittags:", "sonnig,", 17, "Grad")
6 print()
7 print("Schönes Wochenende!")
```

Terminal

```
Wetterbericht für Freitag
------------------------
Morgens: Nebel, 8 Grad
Mittags: sonnig, 17 Grad

Schönes Wochenende!
```

- Jedes `print` schreibt **eine Zeile**. Die nächste Ausgabe beginnt in einer neuen Zeile.
- Du kannst `print` mehrere Werte geben, durch Kommas getrennt (Zeile 4 und 5). In der Ausgabe stehen sie dann mit je **einem Leerzeichen** dazwischen. Deshalb steht das Komma hinter „Nebel" im Text selbst: `"Nebel,"`.
- Zahlen wie `8` und `17` brauchen keine Anführungszeichen. `print` schreibt sie einfach hin.
- `print()` ohne etwas in den Klammern (Zeile 6) gibt eine leere Zeile aus.

Konsole oder Display? `print` schreibt **nicht** auf den Bildschirm des PicoBoy, sondern ins Terminal. Beides kann man in einem Programm mischen:

k03_konsole_display.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Schau ins", 50, 100, pbc.WHITE)
5 pbc.text24("Terminal!", 50, 130, pbc.YELLOW)
6
7 print("Hier ist das Terminal.")
8 print("Umlaute gehen hier: ä ö ü ß")
```



Terminal

```
Hier ist das Terminal.
Umlaute gehen hier: ä ö ü ß
```

Im Terminal funktionieren auch Umlaute und ß, auf dem Display nicht. Für längere Texte, Tabellen und Zwischenergebnisse ist die Konsole deshalb oft praktischer als das Display. Alte Ausgaben löschst du mit **Konsole leeren** über dem Terminal.

3.2 Eingaben mit `input`

Mit `input` stellt dein Programm eine Frage und **wartet**, bis jemand im Terminal eine Antwort eintippt und **Enter** drückt:

k03_input.py

```
1 name = input("Wie heißt du? ")
2 print("Hallo,", name)
3 print("Schön, dass du da bist!")
```

Terminal (Eingabe: Mia)

```
Wie heißt du? Mia
Hallo, Mia
Schön, dass du da bist!
```

So läuft das Programm ab:

1. Zeile 1: `input("Wie heißt du? ")` schreibt die Frage ins Terminal und wartet. Klicke ins Terminal, tippe deinen Namen und drücke **Enter**.
2. Was du getippt hast, gibt `input` an das Programm zurück. Mit `name = …` merkt sich das Programm die Antwort unter dem Namen `name`. So einen Namen für einen Wert nennt man **Variable**. In Kapitel 4 lernst du Variablen genauer kennen.
3. Zeile 2: `print("Hallo,", name)` schreibt „Hallo,“ und danach den Wert von `name`. Hier steht `name` **ohne** Anführungszeichen. Sonst würde Python das Wort „name“ ausgeben und nicht die Antwort.

Das Leerzeichen am Ende von `"Wie heißt du? "` sorgt dafür, dass deine Antwort nicht direkt am Fragezeichen klebt.

> Solange das Programm auf eine Eingabe wartet, passiert sonst nichts. Wenn dein Programm „hängt“, schau ins Terminal: Vielleicht wartet es auf dich. Mit ■ **Stop** brichst du ab.

Die Antwort kannst du natürlich auch auf dem Display zeigen:

k03_input_display.py

```
1 import pbc
2
3 tier = input("Was ist dein Lieblingstier? ")
4
5 pbc.clear(pbc.BLUE)
6 pbc.text18("Lieblingstier:", 20, 100, pbc.WHITE)
7 pbc.text24(tier, 20, 130, pbc.YELLOW)
```



Terminal (Eingabe: Pinguin)

```
Was ist dein Lieblingstier? Pinguin
```

In Zeile 7 steht `tier` an der Stelle, an der sonst ein Text in Anführungszeichen steht. `pbc.text24` schreibt dann die Antwort auf den Bildschirm. Tippst du ein Tier mit Umlaut ein, etwa „Möwe“, siehst du auf dem Display wieder die Fragezeichen.

3.3 Texte und Zahlen: `int()`, `float()`, `str()`

Was gibt dieses Programm aus, wenn du `21` eingibst?

k03_text_plus.py

```
1 zahl = input("Gib eine Zahl ein: ")
2 print("Das Doppelte:", zahl + zahl)
```

Terminal (Eingabe: 21)

```
Gib eine Zahl ein: 21
Das Doppelte: 2121
```

Nicht 42, sondern 2121! Der Grund: `input` liefert **immer einen Text**, auch wenn du Ziffern eintippst. Für Python ist die Antwort also `"21"`, nicht `21`. Und `+` bei zwei Texten hängt sie aneinander: `"21" + "21"` ergibt `"2121"`.

Um mit der Antwort zu rechnen, wandelst du den Text mit `int()` in eine **ganze Zahl** um (*int* von englisch *integer*, ganze Zahl):

k03_int.py

```
1 zahl = int(input("Gib eine Zahl ein: "))
2 print("Das Doppelte:", zahl + zahl)
```

Terminal (Eingabe: 21)

```
Gib eine Zahl ein: 21
Das Doppelte: 42
```

`int(input("…"))` liest man von innen nach außen: Erst stellt `input` die Frage und liefert den Text `"21"`, dann macht `int` daraus die Zahl `21`. Jetzt rechnet `+` richtig. Mehr übers Rechnen erfährst du in Kapitel 4.

Text und Zahl lassen sich nicht addieren. Vergisst du `int`, versucht Python, zu einem Text eine Zahl zu addieren:

k03_fehler_text_zahl.py

```
1 alter = input("Wie alt bist du? ")
2 print("Nächstes Jahr bist du", alter + 1)
```

Terminal (Eingabe: 13)

```
Wie alt bist du? 13

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported types for __add__: 'str', 'int'
```

`TypeError` heißt: Die Werte haben den falschen **Typ**. `'str'` steht für einen Text, `'int'` für eine ganze Zahl. Lösung: `int(input(…))`.

Mit `int` läuft das Programm. Aber `int` kann nur Texte umwandeln, die wirklich wie eine ganze Zahl aussehen. Bei `dreizehn` gibt Python auf:

k03_fehler_int.py

```
1 alter = int(input("Wie alt bist du? "))
2 print("Nächstes Jahr bist du", alter + 1)
```

Terminal (Eingabe: dreizehn)

```
Wie alt bist du? dreizehn

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid syntax for integer with base 10: 'dreizehn'
```

`ValueError` bedeutet: Der Typ stimmt (ein Text), aber der **Wert** passt nicht.

Kommazahlen mit `float()`. Für Zahlen mit Nachkommastellen gibt es `float()`:

k03_float.py

```
1 preis = float(input("Was kostet die Limo in Euro? "))
2 print("Mit Pfand kostet sie", preis + 0.25, "Euro.")
```

Terminal (Eingabe: 1.5)

```
Was kostet die Limo in Euro? 1.5
Mit Pfand kostet sie 1.75 Euro.
```

Python schreibt Kommazahlen mit Punkt. Statt 1,5 tippst du `1.5`. Mit einem Komma klappt die Umwandlung nicht:

Terminal (Eingabe: 1,5)

```
Was kostet die Limo in Euro? 1,5

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid syntax for number
```

Zahlen in Text verwandeln mit `str()`. Manchmal brauchst du den umgekehrten Weg: aus einer Zahl einen Text machen. Das tut `str()` (von *string*):

k03_str.py

```
1 import pbc
2
3 jahr = int(input("In welchem Jahr bist du geboren? "))
4 print("Mit 18 ist man volljährig.")
5 print("Du wirst es im Jahr " + str(jahr + 18) + ".")
6
7 pbc.clear()
8 pbc.text18("Volljaehrig im Jahr", 20, 100, pbc.WHITE)
9 pbc.text24(str(jahr + 18), 20, 130, pbc.GREEN)
```



Terminal (Eingabe: 2012)

```
In welchem Jahr bist du geboren? 2012
Mit 18 ist man volljährig.
Du wirst es im Jahr 2030.
```

- In Zeile 5 werden drei Texte mit `+` zu einem Satz zusammengeklebt. Das geht nur, wenn alle Teile Texte sind. Deshalb macht `str(jahr + 18)` aus der Zahl 2030 den Text `"2030"`. Bei `+` fügt Python übrigens keine Leerzeichen ein, die stehen deshalb in den Texten selbst.
- Die Textbefehle für das Display erwarten einen Text. Deshalb steht auch in Zeile 9 `str(…)`.

Das Display braucht einen Text. Steht in Zeile 5 eines Programms `pbc.text24(punkte, 20, 120, pbc.WHITE)` und ist `punkte` eine Zahl, bricht das Programm ab:

```
Terminal
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError: can't convert 'int' object to str implicitly
```

Python wandelt die Zahl nicht von selbst um („implicitly“). Lösung: `pbc.text24(str(punkte), 20, 120, pbc.WHITE)`.

3.4 Die REPL: Python direkt ausprobieren (am Gerät)

Bisher hast du immer ein ganzes Programm geschrieben und gestartet. Wenn du nur schnell etwas ausprobieren willst, etwa „Was ergibt `"3" + "4"`?“, geht das auch einfacher: mit der **REPL**. Das steht für *Read – Eval – Print – Loop*: Python **liest** eine Zeile, **wertet** sie aus, **zeigt** das Ergebnis und **wiederholt** das Ganze.

Die REPL gibt es nur auf dem **echten Gerät**:

1. Schließe den PicoBoy an, wähle als Ziel **PBC+** und klicke auf **Verbinden**. Auf dem Display des PicoBoy erscheint „REPL“.
2. Klicke ins Terminal und drücke **Enter**. Es erscheint die Eingabeaufforderung `>>>`. Jetzt wartet Python auf dich.
3. Tippe eine Zeile Python ein und drücke **Enter**.

So kann das aussehen:

```
>>> 3 + 4
7
>>> "3" + "4"
'34'
>>> int("3") + int("4")
7
>>> print("Hallo!")
Hallo!
>>> name = "Mia"
>>> name
'Mia'
>>> import pbc
>>> pbc.clear(pbc.RED)
>>>
```

- Steht hinter `>>>` ein Wert oder eine Rechnung, zeigt die REPL das Ergebnis in der nächsten Zeile an, ganz ohne `print`.
- Texte zeigt sie dabei in **einfachen** Anführungszeichen, `'34'`. So siehst du sofort, dass das Ergebnis ein Text ist und keine Zahl.

- Hinter `name = "Mia"` kommt keine Ausgabe: Python hat sich den Wert nur gemerkt. Tippst du danach `name`, zeigt die REPL ihn an.
- Auch PicoBoy-Befehle funktionieren. Nach `pbc.clear(pbc.RED)` wird das Display sofort rot.

Mit der Pfeiltaste ↑ holst du die letzte Zeile zurück und kannst sie ändern. Was du in der REPL tippst, wird nicht gespeichert. Für alles, was du behalten willst, schreibst du ein Programm im Editor.

> **Im Emulator gibt es keine REPL.** Dort nimmt das Terminal nur Eingaben an, während ein Programm mit `input` auf eine Antwort wartet. Ohne Gerät probierst du kleine Dinge deshalb mit einem kurzen Programm und `print` aus, zum Beispiel `print("3" + "4")`.

Ausprobieren: Das Begrüßungsprogramm

Das Programm stellt drei Fragen, antwortet in der Konsole und begrüßt dich auf dem Display:

k03_begruessung.py

```python
# Das Begrüßungsprogramm
import pbc

print("Willkommen beim PicoBoy!")
print("Bitte beantworte drei Fragen.")
print()

# Fragen stellen
name = input("Wie heißt du? ")
ort = input("Wo wohnst du? ")
alter = int(input("Wie alt bist du? "))

# Antwort in der Konsole
print()
print("Danke, " + name + "!")
print("Du wohnst in", ort, "und bist", alter, "Jahre alt.")
print("In zehn Jahren bist du", alter + 10, "Jahre alt.")

# Begrüßung auf dem Display
pbc.clear(pbc.BLUE)
pbc.text18("Willkommen,", 20, 80, pbc.WHITE)
pbc.text24(name, 20, 110, pbc.YELLOW)
pbc.text12("aus " + ort, 20, 150, pbc.WHITE)
pbc.text12("Alter: " + str(alter), 20, 170, pbc.WHITE)
```



Terminal (Eingaben: Lena, Kassel, 13)

```
Willkommen beim PicoBoy!
Bitte beantworte drei Fragen.

Wie heißt du? Lena
Wo wohnst du? Kassel
Wie alt bist du? 13

Danke, Lena!
Du wohnst in Kassel und bist 13 Jahre alt.
In zehn Jahren bist du 23 Jahre alt.
```

- Zeile 4 bis 6 erklären in der Konsole, was gleich passiert. Das leere `print()` macht etwas Abstand.
- Zeile 9 bis 11 stellen die Fragen. Name und Wohnort bleiben Texte, das Alter wird mit `int` zur Zahl, weil in Zeile 17 damit gerechnet wird.
- In Zeile 15 klebt `+` den Text zusammen, damit das Ausrufezeichen direkt hinter dem Namen steht. Mit einem Komma stünde ein Leerzeichen davor.
- Zeile 16 und 17 mischen Texte und Zahlen. Bei `print` mit Kommas ist das erlaubt, `print` wandelt die Zahlen selbst um.
- Zeile 23 und 24: Für das Display werden die Teile mit `+` zu **einem** Text zusammengesetzt. Das Alter muss dafür mit `str` wieder zum Text werden.

Zum Verändern: Frag zusätzlich nach dem Lieblingsessen und zeige es auf dem Display. Was passiert, wenn du als Namen „Jürgen“ eingibst? Was, wenn du beim Alter „dreizehn“ schreibst?

Übungsaufgaben

Aufgabe 1 ⭐ Was steht im Terminal? Schreibe auf, was dieses Programm ausgibt. Achte auf jedes Leerzeichen. Prüfe danach im Emulator.

k03_aufgabe1.py

```
1 print("Eins", "Zwei")
2 print("Drei" + "Vier")
3 print()
4 print("Fünf", 6, "Sieben" + "8")
5 print("1" + "2", 1, 2)
```

Aufgabe 2 ⭐ Ein Steckbrief Schreibe ein Programm, das in der Konsole einen Steckbrief für dein Lieblingstier ausgibt: eine Überschrift mit einer Linie aus `=` darüber und darunter, dann mindestens vier Angaben wie Lebensraum, Größe und Futter. Benutze `print` mit mehreren Werten und mindestens eine leere Zeile.

Aufgabe 3 ⭐⭐ Finde den Fehler Das Programm soll das nächste Jahr ausrechnen. Teste es mit `2026`. Lies die Fehlermeldung und korrigiere das Programm.

k03_aufgabe3.py

```
1 jahr = input("Welches Jahr haben wir? ")
2 print("Nächstes Jahr ist", jahr + 1)
```

Aufgabe 4 ⭐⭐ Ein Schild zum Eintippen Schreibe ein Programm, das fragt: „Was soll auf dem Display stehen?“ Die Antwort erscheint groß auf dem Display. In der Konsole bestätigt das Programm, was angezeigt wird.

Aufgabe 5 ⭐⭐ Die Kinokarte Frag nach einem Film, der Uhrzeit und der Nummer des Saals (als Zahl mit `int`). Gib die Angaben in einer Zeile in der Konsole aus und zeige auf weißem Hintergrund eine Kinokarte mit „Beginn: … Uhr“ und „Saal …“.

Aufgabe 6 ⭐⭐⭐ Text oder Zahl? Überlege zuerst, was herauskommt. Prüfe dann in der REPL am Gerät oder mit `print(…)` in einem Programm im Emulator: `"3" + "4"`, `3 + 4`, `int("3") + int("4")`, `str(3) + str(4)`, `float("3") + 4`.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `print(…)` schreibt eine Zeile ins Terminal. Mehrere Werte trennst du mit Kommas, sie erscheinen mit Leerzeichen dazwischen. `print()` macht eine leere Zeile.
- `antwort = input("Frage? ")` zeigt die Frage, wartet auf eine Eingabe und merkt sich die Antwort unter dem Namen `antwort`.
- `input` liefert **immer Text**. `int(…)` macht daraus eine ganze Zahl, `float(…)` eine Kommazahl (mit Punkt: `1.5`).
- `str(…)` macht aus einer Zahl einen Text, etwa für `+` mit Texten oder für das Display.
- `+` addiert Zahlen, aber hängt Texte aneinander. Text + Zahl ergibt einen `TypeError`.
- Die REPL (`>>>`) gibt es am verbundenen Gerät im Terminal. Dort probierst du einzelne Zeilen Python sofort aus.

KAPITEL 4

Variablen und Rechnen

Worum geht's? Programme müssen sich Dinge merken: einen Punktestand, einen Namen, ein Messergebnis. Dafür gibt es Variablen. Du lernst, mit Variablen zu rechnen, sie zu verändern, Zufallszahlen zu würfeln und Ergebnisse ordentlich als Text auszugeben.

4.1 Variablen: Namen für Werte

Eine **Variable** ist ein Name für einen Wert. Du kannst dir eine Variable wie ein beschriftetes Kästchen vorstellen: Auf dem Etikett steht der Name, darin liegt der Wert. In Kapitel 3 hast du schon mit `name = input(…)` eine Variable angelegt.

k04_variablen.py

```
1 spieler = "Ole"
2 punkte = 120
3 leben = 3
4 print(spieler, "hat", punkte, "Punkte und", leben, "Leben.")
5
6 punkte = 150
7 print("Jetzt hat", spieler, punkte, "Punkte.")
```

Terminal

```
Ole hat 120 Punkte und 3 Leben.
Jetzt hat Ole 150 Punkte.
```

- `punkte = 120` heißt: „Lege den Wert 120 unter dem Namen `punkte` ab." Das Gleichheitszeichen ist hier **kein** Vergleich wie in Mathe, sondern eine **Zuweisung**: Der Wert rechts kommt in die Variable links.
- Die Zeilen 1 bis 3 legen drei Variablen an. In Zeile 4 setzt Python überall, wo ein Variablenname steht, den Wert der Variablen ein.
- Zeile 6 weist `punkte` einen neuen Wert zu. Der alte Wert 120 ist damit weg. Eine Variable enthält immer nur **einen** Wert, nämlich den zuletzt zugewiesenen.

Regeln für Namen:

- Erlaubt sind Buchstaben, Ziffern und der Unterstrich `_`. Ein Name darf aber **nicht mit einer Ziffer beginnen**: `spieler2` geht, `2spieler` nicht.
- Keine Leerzeichen: statt `hohe punkte` schreibst du `hohe_punkte`.
- Groß- und Kleinschreibung zählt: `Punkte` und `punkte` sind zwei verschiedene Variablen.
- Wörter, die Python selbst benutzt, wie `import`, `if` oder `for`, gehen nicht.
- In diesem Buch schreiben wir Namen klein, ohne Umlaute und ß (`groesse`, `uebrig`) und so, dass man versteht, was drinsteht: `punkte` ist besser als `p`.

Eine Variable muss es geben, bevor du sie benutzt. Ein Tippfehler im Namen fällt erst auf, wenn die Zeile ausgeführt wird:

k04_fehler_name.py

```
1 punkte = 120
2 print("Punkte:", punkt)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'punkt' isn't defined
```

`punkt` ist nicht `punkte`. Python kennt den Namen nicht und meldet einen `NameError`. Ein Name, der mit einer Ziffer beginnt (`2spieler = "Ole"`), führt dagegen sofort zu einem `SyntaxError`.

4.2 Ganze Zahlen, Kommazahlen, Texte – Datentypen

Jeder Wert hat einen **Datentyp**. Er bestimmt, was man mit dem Wert machen kann. Mit `type(…)` fragst du Python nach dem Typ:

k04_typen.py

```
1 print(type(14))
2 print(type(1.62))
3 print(type("Jonas"))
4 print(type("14"))
```

Terminal

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'str'>
```

Typ	Bedeutung	Beispiele
`int`	ganze Zahl (*integer*)	`14`, `0`, `-5`, `2026`
`float`	Kommazahl (*floating point*), mit Punkt geschrieben	`1.62`, `0.5`, `-3.0`
`str`	Text (*string*), in Anführungszeichen	`"Jonas"`, `"14"`, `""`

Achte auf die letzte Zeile: `"14"` ist ein **Text**, weil es in Anführungszeichen steht, auch wenn nur Ziffern darin sind. Genau so einen Text liefert `input`. Umwandeln kannst du mit `int()`, `float()` und `str()` (Kapitel 3).

Einen vierten Typ lernst du in Kapitel 7 kennen: `bool` für „wahr“ und „falsch“.

4.3 Rechnen: `+ - * /`, `//`, `%`, `**`

Python kann rechnen wie ein Taschenrechner. Statt Zahlen kannst du auch Variablen einsetzen:

k04_rechnen.py

```
1 a = 17
2 b = 5
3 print("a + b  =", a + b)
4 print("a - b  =", a - b)
5 print("a * b  =", a * b)
6 print("a / b  =", a / b)
7 print("a // b =", a // b)
8 print("a % b  =", a % b)
9 print("a ** 2 =", a ** 2)
```

Die Ergebnisse aus dem Terminal stehen in der letzten Spalte der Tabelle:

Zeichen	Rechenart	Beispiel	Ergebnis
`+`	plus	`17 + 5`	`22`
`-`	minus	`17 - 5`	`12`
`*`	mal	`17 * 5`	`85`
`/`	geteilt (Ergebnis immer `float`)	`17 / 5`	`3.4`
`//`	ganzzahlig geteilt (Nachkommastellen fallen weg)	`17 // 5`	`3`
`%`	Rest beim Teilen („modulo“)	`17 % 5`	`2`
`**`	hoch	`17 ** 2`	`289`

Reihenfolge: Es gilt wie in Mathe „Punkt vor Strich“. `**` kommt noch vor `*` und `/`. Mit Klammern bestimmst du selbst, was zuerst gerechnet wird:

k04_reihenfolge.py

```
1 print(2 + 3 * 4)
2 print((2 + 3) * 4)
3 print(10 / 2)
4 print(0.1 + 0.2)
5 print("Ha" * 3)
```

Terminal

```
14
20
5.0
0.30000000000000004
HaHaHa
```

- `10 / 2` ergibt `5.0` und nicht `5`: Das Ergebnis von `/` ist immer eine Kommazahl. Rechnest du mit einer Kommazahl, kommt auch eine heraus.

- `0.1 + 0.2` ergibt nicht genau `0.3`. Der Computer speichert Kommazahlen im Zweiersystem, und darin lässt sich 0,1 nicht genau darstellen. Die winzige Abweichung ist normal. Wie du solche Zahlen schön gerundet ausgibst, steht in 4.6.
- `"Ha" * 3` wiederholt einen Text: `HaHaHa`.

Ganzzahlig teilen und Rest. `//` und `%` gehören zusammen. 135 Minuten sind 2 ganze Stunden (`135 // 60`), und es bleiben 15 Minuten übrig (`135 % 60`):

k04_rest.py

```
# Wie lang ist der Film?
minuten = 135
stunden = minuten // 60
rest = minuten % 60
print(minuten, "Minuten sind", stunden, "Stunden und", rest, "Minuten.")

# Ist die Zahl gerade?
zahl = 27
print("27 % 2 =", zahl % 2)
```

Terminal

```
135 Minuten sind 2 Stunden und 15 Minuten.
27 % 2 = 1
```

Mit `% 2` prüfst du, ob eine Zahl gerade ist: Bei geraden Zahlen ist der Rest `0`, bei ungeraden `1`.

Durch null teilen geht nicht, auch nicht mit `//` oder `%`:

k04_fehler_null.py

```
kuchenstuecke = 12
gaeste = 0
print("Jeder Gast bekommt", kuchenstuecke / gaeste, "Stücke.")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: divide by zero
```

4.4 Variablen verändern: `x = x + 1` und `x += 1`

In einem Spiel ändert sich der Punktestand ständig. Dazu rechnest du mit dem alten Wert und speicherst das Ergebnis wieder in derselben Variable:

k04_veraendern.py

```
punkte = 0
print("Start:", punkte)

punkte = punkte + 10
print("Münze gesammelt:", punkte)
```

```
 7 punkte += 50
 8 print("Stern gesammelt:", punkte)
 9
10 punkte -= 5
11 print("Gegner berührt:", punkte)
12
13 punkte *= 2
14 print("Bonusrunde:", punkte)
```

Terminal

```
Start: 0
Münze gesammelt: 10
Stern gesammelt: 60
Gegner berührt: 55
Bonusrunde: 110
```

`punkte = punkte + 10` wäre in Mathe Unsinn. In Python bedeutet es: Rechne zuerst die **rechte Seite** aus (alter Wert plus 10), dann lege das Ergebnis in `punkte` ab. Weil das so oft vorkommt, gibt es eine Kurzschreibweise:

```
punkte += 50    # dasselbe wie: punkte = punkte + 50
punkte -= 5     # dasselbe wie: punkte = punkte - 5
punkte *= 2     # dasselbe wie: punkte = punkte * 2
```

Ebenso gibt es `/=`. Die Variable muss vorher einen Wert haben. Deshalb steht in Zeile 1 `punkte = 0`. Ohne diese Zeile gäbe es einen `NameError`.

4.5 Zufallszahlen mit `random`

Für Spiele braucht man den Zufall: einen Würfel, eine zufällige Position, eine Überraschung. Die Befehle dafür stecken im Modul `random`. Wie `pbc` muss es oben im Programm importiert werden:

k04_zufall.py

```
 1 import pbc
 2 import random
 3
 4 wurf = random.randint(1, 6)
 5 print("Gewürfelt:", wurf)
 6
 7 pbc.clear(pbc.WHITE)
 8 pbc.text24("Dein Wurf:", 60, 80, pbc.BLACK)
 9 pbc.text24(str(wurf), 112, 130, pbc.RED)
```



Terminal (Beispiel)

```
Gewürfelt: 4
```

- `random.randint(1, 6)` liefert eine zufällige ganze Zahl von 1 bis 6. Beide Grenzen sind möglich, also auch 1 und 6.
- Bei jedem Testen kommt eine andere Zahl heraus. Dein Ergebnis ist also wahrscheinlich nicht dasselbe wie hier.

- In Zeile 9 macht `str(wurf)` aus der Zahl einen Text für das Display.

Daneben gibt es `random.random()`. Es liefert eine zufällige Kommazahl zwischen 0 und 1, zum Beispiel `0.8140331212901049`. `random.randint` funktioniert auch mit negativen Zahlen: `random.randint(-5, 30)` ist eine gute Zufallstemperatur.

4.6 Texte mit Werten: f-Strings

Texte und Zahlen mit Kommas in `print` oder mit `+` und `str()` zusammenzusetzen, wird schnell unübersichtlich. Einfacher geht es mit einem **f-String**: einem Text mit einem `f` vor dem Anführungszeichen. Was im Text in geschweiften Klammern `{…}` steht, ersetzt Python durch seinen Wert:

k04_fstring.py

```
1 name = "Aylin"
2 punkte = 87
3 maximal = 100
4 print(f"{name} hat {punkte} von {maximal} Punkten.")
5 print(f"Es fehlen {maximal - punkte} Punkte.")
6
7 preis = 1.3
8 print(f"3 Kugeln Eis kosten {3 * preis} Euro.")
9 print(f"3 Kugeln Eis kosten {3 * preis:.2f} Euro.")
```

Terminal

```
Aylin hat 87 von 100 Punkten.
Es fehlen 13 Punkte.
3 Kugeln Eis kosten 3.9000000000000004 Euro.
3 Kugeln Eis kosten 3.90 Euro.
```

- In Zeile 4 werden `{name}`, `{punkte}` und `{maximal}` durch die Werte der Variablen ersetzt. Leerzeichen und Satzzeichen stehen genau da, wo du sie hinschreibst.
- In den Klammern darf auch eine Rechnung stehen (Zeile 5 und 8).
- Zeile 8 zeigt wieder die winzige Ungenauigkeit von Kommazahlen. In Zeile 9 steht hinter der Rechnung `:.2f`. Das bedeutet: als Kommazahl mit **2** Nachkommastellen ausgeben, gerundet. `:.1f` gibt eine Nachkommastelle, `:.0f` keine.
- Vergisst du das `f`, erscheinen die Klammern einfach mit im Text: `print("{punkte} Punkte")` zeigt `{punkte} Punkte`.

Die geschweiften Klammern findest du auf der Tastatur mit **AltGr + 7** und **AltGr + 0**.

Ein f-String ist ein ganz normaler Text. Deshalb kannst du ihn auch direkt an die Textbefehle des Displays geben, ein `str()` ist nicht nötig:

k04_fstring_display.py

```
 1 import pbc
 2
 3 # 100-Meter-Lauf
 4 strecke = 100
 5 zeit = 14.2
 6 tempo = strecke / zeit
 7
 8 pbc.clear()
 9 pbc.text18("100-Meter-Lauf", 20, 60, pbc.WHITE)
10 pbc.text18(f"Zeit: {zeit} s", 20, 110, pbc.YELLOW)
11 pbc.text18(f"Tempo: {tempo:.1f} m/s", 20, 140, pbc.GREEN)
12 pbc.text18(f"= {tempo * 3.6:.1f} km/h", 20, 170, pbc.GREEN)
```



Zeile 6 rechnet die Geschwindigkeit in Metern pro Sekunde aus. Zeile 11 und 12 geben sie mit einer Nachkommastelle aus, einmal in m/s und einmal umgerechnet in km/h (mal 3,6).

Ausprobieren: Der Taschenrechner

Der Taschenrechner fragt nach zwei Zahlen und rechnet alles aus, was er kann:

k04_taschenrechner.py

```
 1 # Der Taschenrechner
 2 print("Taschenrechner")
 3 a = float(input("Erste Zahl: "))
 4 b = float(input("Zweite Zahl: "))
 5
 6 summe = a + b
 7 differenz = a - b
 8 produkt = a * b
 9 quotient = a / b
10
11 print(f"{a} + {b} = {summe}")
12 print(f"{a} - {b} = {differenz}")
13 print(f"{a} * {b} = {produkt}")
14 print(f"{a} / {b} = {quotient:.3f}")
15 print(f"{a} hoch {b} = {a ** b}")
```

Terminal (Eingaben: 7 und 2)

```
Taschenrechner
Erste Zahl: 7
Zweite Zahl: 2
7.0 + 2.0 = 9.0
7.0 - 2.0 = 5.0
7.0 * 2.0 = 14.0
7.0 / 2.0 = 3.500
7.0 hoch 2.0 = 49.0
```

- Zeile 3 und 4 wandeln die Eingaben mit `float` um. So kann man auch Kommazahlen wie `2.5` eingeben. Deshalb erscheint `7` als `7.0`.
- Zeile 6 bis 9 rechnen die Ergebnisse aus und merken sie sich in Variablen. Zeile 11 bis 15 setzen sie mit f-Strings in die Ausgabe ein.
- In Zeile 14 sorgt `:.3f` für drei Nachkommastellen. Probiere 1 und 3 als Eingaben, dann siehst du, warum das sinnvoll ist.

Zum Verändern: Gib alle Ergebnisse mit zwei Nachkommastellen aus. Ergänze `//` und `%`. Zeige die Ergebnisse zusätzlich auf dem Display. Was passiert, wenn die zweite Zahl `0` ist? (Wie man das abfängt, lernst du in Kapitel 7.)

Übungsaufgaben

Aufgabe 1 ⭐ Was gibt das Programm aus? Überlege Zeile für Zeile, welchen Wert die Variablen haben. Schreibe die Ausgabe auf und prüfe sie danach im Emulator.

k04_aufgabe1.py

```
1 x = 7
2 y = x * 2
3 x = x + 1
4 print(x, y)
5 print(17 // 5, 17 % 5, 2 ** 3)
6 print(9 / 3, "ab" * 2)
```

Aufgabe 2 ⭐⭐ Finde den Fehler Das Programm meldet keinen Fehler, rechnet aber falsch. Teste es mit dem Preis `1.5`. Was kommt heraus, und warum? Korrigiere es und gib den Preis mit zwei Nachkommastellen aus.

k04_aufgabe2.py

```
1 preis = input("Preis pro Heft in Euro: ")
2 anzahl = 4
3 gesamt = preis * anzahl
4 print(f"{anzahl} Hefte kosten {gesamt} Euro.")
```

Aufgabe 3 ⭐⭐ Sekunden umrechnen Eine Anzahl Sekunden wird eingegeben. Das Programm gibt aus, wie viele Stunden, Minuten und Sekunden das sind, zum Beispiel `5000 s = 1 h 23 min 20 s`. Tipp: Eine Stunde hat 3600 Sekunden. Benutze `//` und `%`.

Aufgabe 4 ⭐⭐ Zwei Würfel Würfle zwei Würfel und zeige auf dem Display beide Augenzahlen und ihre Summe, zum Beispiel `4 + 3` und darunter `= 7`.

Aufgabe 5 ⭐⭐⭐ Pizza-Party Eine Pizza hat 8 Stücke. Das Programm fragt, wie viele Personen kommen und wie viele Stücke jede Person isst. Es gibt aus, wie viele Stücke gebraucht werden, wie viele Pizzen man bestellen muss und wie viele Stücke übrig bleiben. Schwierig: Die Zahl der Pizzen muss **aufgerundet** werden. Wie geht das nur mit `+` und `//`?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `name = wert` legt einen Wert in einer Variable ab. Rechts wird zuerst ausgerechnet, dann wird zugewiesen.
- Namen: Buchstaben, Ziffern, `_`; nicht mit einer Ziffer beginnen; Groß-/Kleinschreibung zählt.
- Datentypen: `int` (ganze Zahl), `float` (Kommazahl mit Punkt), `str` (Text). `type(…)` verrät den Typ.
- Rechnen: `+ - * /`, `//` (ganzzahlig teilen), `%` (Rest), `**` (hoch); Punkt vor Strich, Klammern zuerst. `/` liefert immer `float`.
- Verändern: `x = x + 1` oder kurz `x += 1` (ebenso `-=`, `*=`, `/=`).
- `import random`, dann `random.randint(a, b)` für eine ganze Zufallszahl von a bis b, `random.random()` für eine Kommazahl zwischen 0 und 1.
- f-Strings: `f"{name} hat {punkte} Punkte"`; `{wert:.2f}` für zwei Nachkommastellen.

KAPITEL 5

Zeichnen und Farben

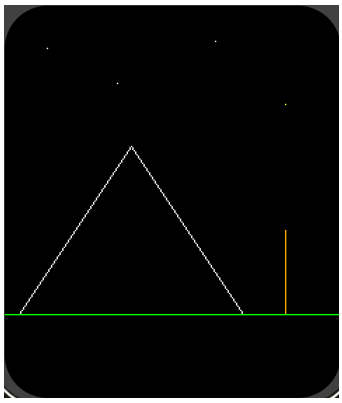
Worum geht's? Bisher hast du nur Text auf den Bildschirm geschrieben. In diesem Kapitel zeichnest du Punkte, Linien, Rechtecke, Kreise und Ellipsen und mischst dir eigene Farben. Am Ende baust du aus diesen Formen ein ganzes Bild.

5.1 Pixel und Linien

Der Bildschirm besteht aus 240 × 280 Pixeln (Abschnitt 2.3). Mit `pbc.pixel` färbst du ein einzelnes Pixel, mit `pbc.line` ziehst du eine Linie zwischen zwei Punkten:

k05_nachthimmel.py

```
 1 import pbc
 2
 3 pbc.clear()
 4 # Einzelne Pixel: Sterne
 5 pbc.pixel(30, 30, pbc.WHITE)
 6 pbc.pixel(80, 55, pbc.WHITE)
 7 pbc.pixel(150, 25, pbc.WHITE)
 8 pbc.pixel(200, 70, pbc.YELLOW)
 9 # Schraege Linien: ein Berg
10 pbc.line(10, 220, 90, 100, pbc.WHITE)
11 pbc.line(90, 100, 170, 220, pbc.WHITE)
12 # Waagrecht und senkrecht
13 pbc.hline(0, 220, 240, pbc.GREEN)
14 pbc.vline(200, 160, 60, pbc.ORANGE)
```

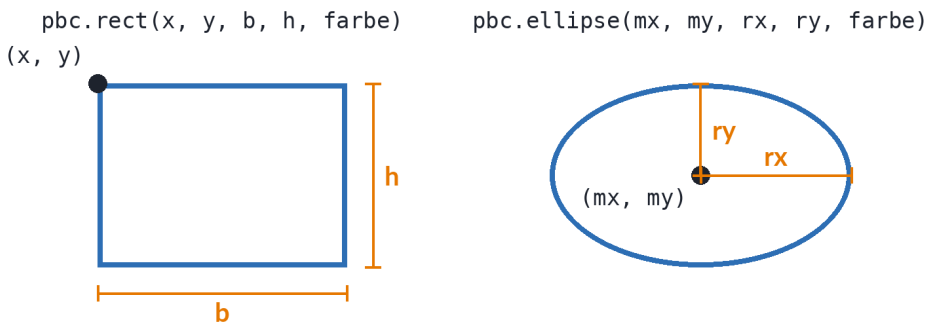


- **Zeile 5–8:** `pbc.pixel(x, y, farbe)` färbt genau das Pixel an der Stelle (x, y). Ein Pixel ist sehr klein, die vier „Sterne" sind kaum zu sehen. Einzelne Pixel braucht man eher für feine Muster als für große Bilder.
- **Zeile 10–11:** `pbc.line(x1, y1, x2, y2, farbe)` zeichnet eine gerade Linie vom Punkt (x1, y1) zum Punkt (x2, y2). Die erste Linie geht von links unten (10, 220) zur Bergspitze (90, 100), die zweite von der Spitze nach rechts unten. Beide Linien treffen sich, weil der Endpunkt der ersten derselbe ist wie der Anfangspunkt der zweiten.
- **Zeile 13:** `pbc.hline(x, y, laenge, farbe)` zeichnet eine **waagrechte** Linie. Sie beginnt bei (x, y) und geht `laenge` Pixel nach rechts. Hier geht sie bei y = 220 über die ganze Breite.
- **Zeile 14:** `pbc.vline(x, y, laenge, farbe)` zeichnet eine **senkrechte** Linie, die bei (x, y) beginnt und nach **unten** geht. Der orange Strich reicht von y = 160 bis zum Boden bei y = 220.

Mit `line` ginge das auch, aber bei `hline` und `vline` gibst du nur die Länge an, ohne den Endpunkt auszurechnen.

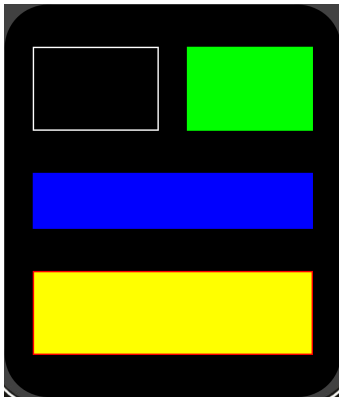
5.2 Rechtecke und Ellipsen, gefüllt und als Umriss

Rechtecke zeichnest du mit `pbc.rect`. Die ersten beiden Zahlen sind die **linke obere Ecke**, dann folgen Breite und Höhe:



k05_rechtecke.py

```
 1 import pbc
 2
 3 pbc.clear()
 4 # Nur der Umriss
 5 pbc.rect(20, 30, 90, 60, pbc.WHITE)
 6 # Gefuellt: zwei Schreibweisen
 7 pbc.rect(130, 30, 90, 60, pbc.GREEN, fill=True)
 8 pbc.fill_rect(20, 120, 200, 40, pbc.BLUE)
 9 # Gefuelltes Rechteck mit Rand: erst fuellen, dann der Rand
10 pbc.fill_rect(20, 190, 200, 60, pbc.YELLOW)
11 pbc.rect(20, 190, 200, 60, pbc.RED)
```

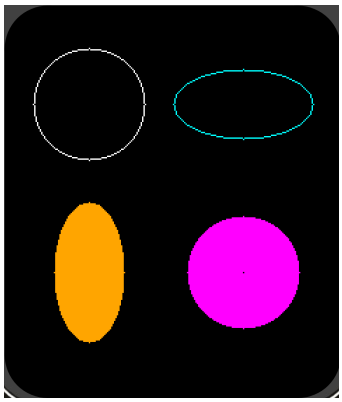


- **Zeile 5:** `pbc.rect(20, 30, 90, 60, pbc.WHITE)` zeichnet nur den **Umriss**: ein Rechteck mit der linken oberen Ecke bei (20, 30), 90 Pixel breit und 60 Pixel hoch.
- **Zeile 7:** Mit `fill=True` wird das Rechteck **gefüllt**. Das ist ein **benanntes Argument**: Du schreibst den Namen `fill`, ein `=` und den Wert `True` (wahr). Benannte Argumente stehen immer am Ende, und man lässt sie weg, wenn man den Standardwert will. Ohne `fill=True` gibt es nur den Umriss.
- **Zeile 8:** `pbc.fill_rect` macht dasselbe wie `pbc.rect(…, fill=True)`, ist nur kürzer zu schreiben. Nimm, was dir besser gefällt.
- **Zeile 10–11:** Erst wird ein gelbes Rechteck gefüllt, dann ein roter Umriss mit denselben Zahlen darübergezeichnet. So bekommt eine Fläche einen Rand.

Ellipsen und Kreise zeichnest du mit `pbc.ellipse`. Hier geben die ersten beiden Zahlen den **Mittelpunkt** an, nicht eine Ecke. Danach folgen zwei **Radien**: `rx` ist der Abstand vom Mittelpunkt zum linken und rechten Rand, `ry` der Abstand zum oberen und unteren Rand. Sind beide gleich, entsteht ein Kreis.

k05_ellipsen.py

```
 1 import pbc
 2
 3 pbc.clear()
 4 # Kreis: beide Radien gleich
 5 pbc.ellipse(60, 70, 40, 40, pbc.WHITE)
 6 # Flache Ellipse: breiter als hoch
 7 pbc.ellipse(170, 70, 50, 25, pbc.CYAN)
 8 # Gefuellt
 9 pbc.ellipse(60, 190, 25, 50, pbc.ORANGE, fill=True)
10 pbc.ellipse(170, 190, 40, 40, pbc.MAGENTA, fill=True)
11 # Mittelpunkt sichtbar machen
12 pbc.pixel(170, 190, pbc.BLACK)
```



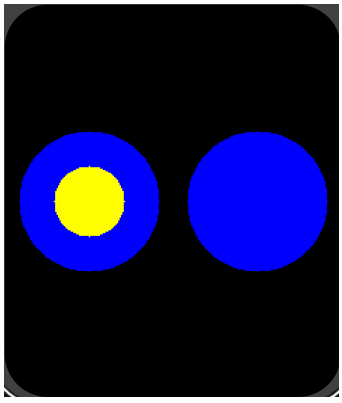
- **Zeile 5:** Ein Kreis um (60, 70) mit Radius 40. Er ist also 80 Pixel breit und reicht von x = 20 bis x = 100.
- **Zeile 7:** `rx = 50`, `ry = 25`: Die Ellipse ist doppelt so breit wie hoch.
- **Zeile 9–10:** Mit `fill=True` wird die Ellipse gefüllt, genau wie beim Rechteck. Die orange Ellipse ist höher als breit, weil `ry` größer ist als `rx`.
- **Zeile 12:** Ein schwarzes Pixel markiert den Mittelpunkt des Magenta-Kreises. Er liegt genau bei den beiden ersten Zahlen.

> **Ecke oder Mitte?** Bei `rect` und `fill_rect` geben x und y die **linke obere Ecke** an, bei `ellipse` den **Mittelpunkt**. Das verwechselt man leicht. Willst du einen Kreis mit Radius 20 in ein Quadrat bei (100, 100) mit 40 Pixeln Seitenlänge setzen, liegt sein Mittelpunkt bei (120, 120).

Was später kommt, liegt oben. Jede Form wird über das gezeichnet, was schon da ist. Die Reihenfolge der Befehle entscheidet also, was man sieht:

k05_reihenfolge.py

```
1 import pbc
2
3 pbc.clear()
4 # Links: erst das grosse, dann das kleine Feld
5 pbc.ellipse(60, 140, 50, 50, pbc.BLUE, fill=True)
6 pbc.ellipse(60, 140, 25, 25, pbc.YELLOW, fill=True)
7 # Rechts: umgekehrt - das kleine Feld wird verdeckt
8 pbc.ellipse(180, 140, 25, 25, pbc.YELLOW, fill=True)
9 pbc.ellipse(180, 140, 50, 50, pbc.BLUE, fill=True)
```



Links wird zuerst der große blaue Kreis gezeichnet und dann der kleine gelbe darauf. Rechts ist es umgekehrt: Der große blaue Kreis kommt zuletzt und deckt den gelben vollständig zu. Der gelbe Kreis wurde zwar gezeichnet, ist aber nicht mehr zu sehen. Zeichne deshalb immer von **hinten nach vorn**: zuerst den Hintergrund, dann große Flächen, zuletzt die kleinen Teile.

Vergisst du die Farbe, bricht das Programm ab. Zeichnungsbefehle brauchen immer **alle** Zahlen und eine Farbe:

```
pbc.fill_rect(20, 20, 100, 60)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: fill_rect() takes 5 positional arguments but 4 were given
```

Ein `TypeError` bedeutet hier: Die Funktion hat nicht die richtige Anzahl an Argumenten bekommen. `fill_rect` erwartet fünf (x, y, Breite, Höhe, Farbe), bekommen hat sie nur vier.

5.3 Farben: die Farbkonstanten und eigene Farben

Die elf fertigen Farben kennst du schon aus Kapitel 2: `pbc.BLACK`, `pbc.WHITE`, `pbc.RED`, `pbc.GREEN`, `pbc.BLUE`, `pbc.YELLOW`, `pbc.CYAN`, `pbc.MAGENTA`, `pbc.ORANGE`, `pbc.PURPLE` und `pbc.GRAY`. Man nennt sie **Konstanten**: Werte mit einem festen Namen, die sich nie ändern. In Python schreibt man Konstanten deshalb mit Großbuchstaben.

Willst du eine andere Farbe, mischst du sie mit `pbc.rgb(r, g, b)`. Die drei Zahlen geben an, wie viel **R**ot, **G**rün und **B**lau in der Farbe steckt, jeweils von 0 (gar nichts) bis 255 (volle Stärke).

k05_farbtabelle.py

```
import pbc

pbc.clear()
# Die drei Grundfarben des Lichts
pbc.fill_rect(15, 15, 45, 24, pbc.rgb(255, 0, 0))
pbc.text12("255, 0, 0  rot", 70, 20, pbc.WHITE)
pbc.fill_rect(15, 46, 45, 24, pbc.rgb(0, 255, 0))
pbc.text12("0, 255, 0  gruen", 70, 51, pbc.WHITE)
pbc.fill_rect(15, 77, 45, 24, pbc.rgb(0, 0, 255))
pbc.text12("0, 0, 255  blau", 70, 82, pbc.WHITE)
# Gemischt
pbc.fill_rect(15, 108, 45, 24, pbc.rgb(255, 255, 0))
pbc.text12("255, 255, 0  gelb", 70, 113, pbc.WHITE)
pbc.fill_rect(15, 139, 45, 24, pbc.rgb(255, 0, 255))
pbc.text12("255, 0, 255  magenta", 70, 144, pbc.WHITE)
pbc.fill_rect(15, 170, 45, 24, pbc.rgb(255, 255, 255))
pbc.text12("255, 255, 255  weiss", 70, 175, pbc.WHITE)
# Halbe Werte und eigene Mischungen
pbc.fill_rect(15, 201, 45, 24, pbc.rgb(128, 128, 128))
pbc.text12("128, 128, 128  grau", 70, 206, pbc.WHITE)
pbc.fill_rect(15, 232, 45, 24, pbc.rgb(120, 70, 20))
pbc.text12("120, 70, 20  braun", 70, 237, pbc.WHITE)
```



Der Bildschirm mischt **Licht**, keine Farbe aus dem Tuschkasten. Deshalb gelten andere Regeln als beim Malen:

- Rot und Grün zusammen ergeben **Gelb** (`255, 255, 0`).
- Alle drei Farben voll ergeben **Weiß** (`255, 255, 255`), keine davon ergibt **Schwarz** (`0, 0, 0`).
- Sind alle drei Werte gleich, entsteht ein **Grau**. Je kleiner die Zahlen, desto dunkler.
- Braun ist ein dunkles Orange: viel Rot, etwas Grün, kaum Blau.

Farben sind Zahlen. Was ist eigentlich `pbc.RED`? Lass es dir mit `print` anzeigen:

k05_farbzahlen.py

```
1 import pbc
2
3 print(pbc.RED)
4 print(pbc.rgb(255, 0, 0))
5 print(pbc.rgb(250, 0, 0))
6
7 # Eine eigene Farbe in einer Variablen merken
8 himmelblau = pbc.rgb(120, 180, 255)
9 print(himmelblau)
10 pbc.clear(himmelblau)
```

Terminal

```
63488
63488
63488
32191
```

- **Zeile 3–4:** `pbc.RED` ist einfach die Zahl 63488. `pbc.rgb(255, 0, 0)` rechnet aus den drei Anteilen genau dieselbe Zahl aus. Für den PicoBoy ist jede Farbe eine Zahl zwischen 0 (Schwarz) und 65535 (Weiß).
- **Zeile 5:** Auch `pbc.rgb(250, 0, 0)` ergibt 63488. Der Bildschirm kann „nur“ 65 536 verschiedene Farben anzeigen. Sehr ähnliche Werte landen deshalb bei derselben Farbe. Mit dem Auge siehst du den Unterschied ohnehin nicht.
- **Zeile 8:** Weil eine Farbe eine Zahl ist, kannst du sie in einer **Variablen** speichern (Kapitel 4). Der Name `himmelblau` sagt mehr als `pbc.rgb(120, 180, 255)`, und wenn du die Farbe ändern willst, änderst du nur diese eine Zeile.
- **Zeile 10:** Die Variable steht überall dort, wo eine Farbe erwartet wird. Der Bildschirm wird himmelblau.

5.4 `pbc.WIDTH` und `pbc.HEIGHT`

Statt 240 und 280 kannst du `pbc.WIDTH` (Breite) und `pbc.HEIGHT` (Höhe) schreiben. Das sind zwei weitere Konstanten des Moduls `pbc`. Damit sieht man im Programm sofort, was eine Zahl bedeutet, und kann damit rechnen, zum Beispiel die Bildschirmmitte:

k05_mitte.py

```
 1 import pbc
 2
 3 mitte_x = pbc.WIDTH // 2
 4 mitte_y = pbc.HEIGHT // 2
 5 print("Mitte:", mitte_x, mitte_y)
 6
 7 pbc.clear()
 8 # Rahmen um den ganzen Bildschirm
 9 pbc.rect(0, 0, pbc.WIDTH, pbc.HEIGHT, pbc.YELLOW)
10 # Kreis genau in der Mitte
11 pbc.ellipse(mitte_x, mitte_y, 50, 50, pbc.RED)
12 # Text zentrieren
13 text = "Mitte"
14 breite = pbc.text24_width(text)
15 pbc.text24(text, mitte_x - breite // 2, 20, pbc.WHITE)
```



Terminal

```
Mitte: 120 140
```

- **Zeile 3–4:** Die Mitte ist die halbe Breite und die halbe Höhe. Wir rechnen mit `//` (ganzzahlig teilen, Kapitel 4), damit eine ganze Zahl herauskommt: 120 und 140.
- **Zeile 9:** Ein Rechteck so groß wie der ganze Bildschirm. Sein Rand liegt genau auf den äußersten Pixeln. An den abgerundeten Ecken des Bildschirms ist er abgeschnitten.
- **Zeile 11:** Weil `ellipse` den Mittelpunkt erwartet, steht der Kreis ohne weitere Rechnung genau in der Mitte.
- **Zeile 13–15:** Einen Text zentrieren: `pbc.text24_width(text)` liefert, wie viele Pixel breit der Text in der Schrift `text24` wird. Der Text muss dann um die halbe Breite links von der Mitte beginnen. Für die anderen Schriften gibt es `pbc.text12_width` und `pbc.text18_width`.

Ganze Zahlen für Koordinaten. `pbc.WIDTH / 2` ergibt `120.0`, eine Kommazahl. Die Zeichenbefehle erwarten aber ganze Zahlen. Steht in Zeile 4 eines Programms `pbc.ellipse(pbc.WIDTH / 2, pbc.HEIGHT / 2, 40, 40, pbc.RED)`, bricht es ab:

Terminal

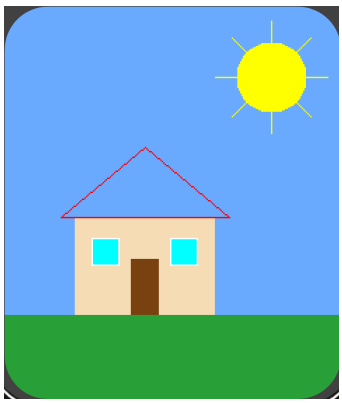
```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: can't convert float to int
```

Rechne Koordinaten deshalb immer mit `//` oder wandle mit `int()` um.

Ausprobieren: Ein Haus mit Sonne

k05_haus.py

```
 1 import pbc
 2
 3 himmel = pbc.rgb(110, 170, 250)
 4 wiese = pbc.rgb(40, 160, 60)
 5 wand = pbc.rgb(240, 220, 180)
 6 haus_x = 50
 7 haus_y = 150
 8
 9 # Hintergrund: Himmel und Wiese
10 pbc.clear(himmel)
11 pbc.fill_rect(0, 220, pbc.WIDTH, pbc.HEIGHT - 220, wiese)
12 # Sonne: erst die Strahlen, dann die Scheibe darueber
13 pbc.line(190, 10, 190, 90, pbc.YELLOW)
14 pbc.line(150, 50, 230, 50, pbc.YELLOW)
15 pbc.line(162, 22, 218, 78, pbc.YELLOW)
16 pbc.line(162, 78, 218, 22, pbc.YELLOW)
17 pbc.ellipse(190, 50, 25, 25, pbc.YELLOW, fill=True)
18 # Haus: Wand, Dach, Tuer, Fenster
19 pbc.fill_rect(haus_x, haus_y, 100, 70, wand)
20 pbc.line(haus_x - 10, haus_y, haus_x + 50, haus_y - 50, pbc.RED)
21 pbc.line(haus_x + 50, haus_y - 50, haus_x + 110, haus_y, pbc.RED)
22 pbc.hline(haus_x - 10, haus_y, 121, pbc.RED)
23 pbc.fill_rect(haus_x + 40, haus_y + 30, 20, 40, pbc.rgb(120, 70, 20))
24 pbc.fill_rect(haus_x + 12, haus_y + 15, 20, 20, pbc.CYAN)
25 pbc.rect(haus_x + 12, haus_y + 15, 20, 20, pbc.WHITE)
26 pbc.fill_rect(haus_x + 68, haus_y + 15, 20, 20, pbc.CYAN)
27 pbc.rect(haus_x + 68, haus_y + 15, 20, 20, pbc.WHITE)
```



- **Zeile 3–5:** Die drei eigenen Farben bekommen sprechende Namen.
- **Zeile 6–7:** `haus_x` und `haus_y` sind die linke obere Ecke der Hauswand. **Alle** Teile des Hauses werden von diesem Punkt aus berechnet: Die Tür beginnt zum Beispiel 40 Pixel weiter rechts und 30 Pixel weiter unten (Zeile 23).
- **Zeile 10–11:** Erst der Hintergrund. Die Wiese beginnt bei y = 220 und ist `pbc.HEIGHT - 220` = 60 Pixel hoch, reicht also genau bis zum unteren Rand.
- **Zeile 13–17:** Vier Linien kreuzen sich im Mittelpunkt der Sonne. Danach deckt die gefüllte Sonnenscheibe die Mitte der Linien zu, nur die Strahlen schauen heraus. Hier nutzen wir absichtlich aus, dass das Spätere oben liegt.
- **Zeile 19–22:** Wand und Dach. Das Dach besteht aus zwei schrägen Linien und einer waagrechten Linie. Es steht auf jeder Seite 10 Pixel über.
- **Zeile 23–27:** Tür und Fenster. Jedes Fenster ist eine cyanfarbene Fläche mit einem weißen Rahmen.

Zum Verändern: Ändere nur `haus_x` und `haus_y`. Das ganze Haus wandert mit, weil alle Teile davon abhängen. Mache aus dem Tag eine Nacht (dunkler Himmel, Mond statt Sonne, gelbe Fenster). Setze einen Schornstein auf das Dach und einen Baum aus einem Rechteck und einer Ellipse daneben.

Übungsaufgaben

Aufgabe 1 ⭐ **Was wird gezeichnet?** Lies das Programm, ohne es zu testen. Skizziere, was auf dem Bildschirm erscheint. Was stellt das Bild dar? Prüfe danach im Emulator.

k05_aufgabe1.py

```
1 import pbc
2
3 pbc.clear(pbc.GRAY)
4 pbc.fill_rect(80, 30, 80, 220, pbc.BLACK)
5 pbc.ellipse(120, 70, 28, 28, pbc.RED, fill=True)
6 pbc.ellipse(120, 140, 28, 28, pbc.YELLOW, fill=True)
7 pbc.ellipse(120, 210, 28, 28, pbc.GREEN, fill=True)
8 pbc.vline(120, 250, 30, pbc.BLACK)
```

Aufgabe 2 ⭐ **Finde die Fehler** Das Programm soll einen Schneemann mit Hut zeichnen, genau in der Mitte des Bildschirms. Es enthält drei Fehler: Zwei melden sich mit einer Fehlermeldung, einer nicht. Finde und korrigiere alle drei.

k05_aufgabe2.py

```
1 import pbc
2
3 pbc.clear(pbc.BLUE)
4 # Drei Kugeln uebereinander, in der Mitte
5 pbc.ellipse(120, 220, 45, 45, pbc.WHITE, fill=True)
6 pbc.ellipse(120, 145, 35, 35, pbc.WHITE, fill=True)
7 pbc.ellipse(120, 85, 25, 25, pbc.WHITE fill=True)
8 # Hut: 40 breit, mittig auf dem Kopf
9 pbc.fill_rect(120, 30, 40, 30, pbc.BLACK)
10 pbc.hline(85, 60, 70)
```

Aufgabe 3 ⭐⭐ **Eine Flagge** Zeichne die Flagge Deutschlands über den ganzen Bildschirm: drei gleich hohe waagrechte Streifen in Schwarz, Rot und Gold. Gold mischst du mit `pbc.rgb(255, 204, 0)`. Rechne die Höhe eines Streifens mit `pbc.HEIGHT` aus, statt sie selbst festzulegen. Reichen die drei Streifen genau bis unten?

Aufgabe 4 ⭐⭐ Die Zielscheibe Zeichne eine Zielscheibe aus fünf gefüllten Kreisen mit demselben Mittelpunkt in der Bildschirmmitte: außen Weiß, dann Schwarz, Blau, Rot und innen Gelb. Jeder Ring ist 20 Pixel breit. In welcher Reihenfolge musst du die Kreise zeichnen und warum?

Aufgabe 5 ⭐⭐⭐ Ein Balkendiagramm Speichere die Höchsttemperaturen von fünf Tagen in Variablen (z. B. `mo = 12`). Zeichne für jeden Tag einen Balken, der auf einer gemeinsamen Grundlinie bei y = 240 steht. Ein Grad soll 8 Pixel hoch sein. Beschrifte die Balken mit den Tagen. Tipp: Weil y nach unten wächst, liegt die obere Kante eines Balkens bei `240 - hoehe`.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `pbc.pixel(x, y, farbe)`, `pbc.line(x1, y1, x2, y2, farbe)`, `pbc.hline(x, y, laenge, farbe)`, `pbc.vline(x, y, laenge, farbe)`.
- `pbc.rect(x, y, b, h, farbe)` zeichnet einen Umriss, mit `fill=True` oder als `pbc.fill_rect` gefüllt. (x, y) ist die **linke obere Ecke**.
- `pbc.ellipse(mx, my, rx, ry, farbe)` zeichnet eine Ellipse um den **Mittelpunkt** (mx, my), mit `rx = ry` einen Kreis, mit `fill=True` gefüllt.
- Was später gezeichnet wird, liegt oben: von hinten nach vorn zeichnen.
- `pbc.rgb(r, g, b)` mischt eine Farbe aus Rot, Grün und Blau (je 0–255). Farben sind Zahlen und lassen sich in Variablen speichern.
- `pbc.WIDTH` und `pbc.HEIGHT` sind 240 und 280. Koordinaten mit `//` berechnen, damit sie ganze Zahlen bleiben.

KAPITEL 6

Schleifen und Zeit

Worum geht's? Ein Programm kann warten und Anweisungen wiederholen, so oft du willst. Mit Schleifen zeichnest du in drei Zeilen hundert Formen, zählst vorwärts und rückwärts und bringst Dinge auf dem Bildschirm in Bewegung. Außerdem erfährst du, warum bewegte Bilder flackern können und was man dagegen tut.

6.1 Warten: `time.sleep` und `time.sleep_ms`

Bisher läuft jedes Programm so schnell es kann von oben nach unten durch. Das Modul `time` enthält Befehle, die das Programm eine Weile **anhalten**:

k06_start.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text24("Auf die Plaetze", 20, 60, pbc.WHITE)
 6 time.sleep(2)
 7 pbc.text24("fertig", 84, 120, pbc.YELLOW)
 8 time.sleep(1.5)
 9 pbc.text24("LOS!", 88, 180, pbc.GREEN)
10 print("Gestartet!")
11 time.sleep_ms(500)
12 print("Eine halbe Sekunde spaeter")
```



Terminal

```
Gestartet!
Eine halbe Sekunde spaeter
```

- **Zeile 2:** Wie `pbc` muss auch das Modul `time` erst importiert werden. Fehlt die Zeile, gibt es einen `NameError`.
- **Zeile 6:** `time.sleep(2)` wartet **2 Sekunden**. Erst danach erscheint „fertig".
- **Zeile 8:** Auch Kommazahlen sind erlaubt: `time.sleep(1.5)` wartet anderthalb Sekunden. Denk an den Punkt statt des Kommas.
- **Zeile 11:** `time.sleep_ms(500)` wartet **500 Millisekunden**, also eine halbe Sekunde. Eine Millisekunde (ms) ist eine tausendstel Sekunde. Für kurze Pausen, etwa in Animationen, ist `sleep_ms` mit ganzen Zahlen bequemer. `pbc.delay(500)` tut übrigens dasselbe.

6.2 Die `for`-Schleife mit `range`

Soll etwas mehrmals passieren, schreibst du es nicht mehrmals hin, sondern benutzt eine **Schleife**:

k06_runden.py

```
1 print("Start")
2 for runde in range(3):
3     print("Runde", runde)
4     print("  laufen ...")
5 print("Ziel!")
```

Terminal

```
Start
Runde 0
  laufen ...
Runde 1
  laufen ...
Runde 2
  laufen ...
Ziel!
```

- **Zeile 2:** `for runde in range(3):` ist der **Kopf** der Schleife. `range(3)` liefert der Reihe nach die Zahlen 0, 1 und 2, also drei Zahlen, beginnend bei **0**. Bei jedem Durchlauf steht die nächste Zahl in der Variablen `runde`. Sie heißt **Schleifenvariable**. Am Ende der Kopfzeile steht ein **Doppelpunkt**.
- **Zeile 3–4:** Diese Zeilen sind um **4 Leerzeichen eingerückt**. Sie bilden den **Schleifenkörper** und werden bei jedem Durchlauf ausgeführt, hier also dreimal.
- **Zeile 5:** Diese Zeile ist nicht mehr eingerückt. Sie gehört nicht zur Schleife und läuft erst danach, einmal.

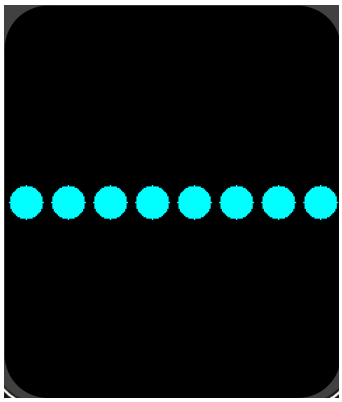
> Nach einer Zeile mit Doppelpunkt rückt der Editor beim **Enter** selbst um 4 Leerzeichen ein. Mit der **Tab-Taste** rückst du um eine Stufe ein, mit **Umschalt + Tab** wieder aus; die **Rücktaste** nimmt in der Einrückung eine ganze Stufe weg.

Spannend wird die Schleife, wenn man mit der Schleifenvariablen rechnet:

k06_perlen.py

```
1 import pbc
2
3 pbc.clear()
4 for i in range(8):
5     pbc.ellipse(15 + i * 30, 140, 12, 12, pbc.CYAN, fill=True)
```

Zeile 5 läuft achtmal. `i` ist nacheinander 0, 1, 2 … 7, der Mittelpunkt `15 + i * 30` also 15, 45, 75 … 225. Jeder Kreis liegt 30 Pixel rechts vom vorigen. Willst du mehr oder weniger Perlen, änderst du nur eine Zahl.



Mit `range(start, ende)` beginnt die Zählung nicht bei 0, sondern bei `start`. Die Zahl `ende` selbst ist **nicht** mehr dabei: `range(1, 6)` liefert 1, 2, 3, 4, 5. Das „bis ausschließlich“ wirkt am Anfang seltsam. Es hat aber einen Vorteil: `range(1, 6)` sind genau `6 - 1` = 5 Zahlen.

Doppelpunkt vergessen: Fehlt in `k06_perlen.py` der Doppelpunkt hinter `for i in range(8)`, kann Python das Programm nicht lesen:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4
SyntaxError: invalid syntax
```

Falsch eingerückt: Alle Zeilen des Schleifenkörpers müssen gleich weit eingerückt sein. Steht eine Zeile weiter rechts als die vorige, meldet Python einen `IndentationError` (Einrückungsfehler):

k06_fehler_einrueckung.py

```
1 for runde in range(3):
2     print("Runde", runde)
3       print("  laufen ...")
```

Terminal

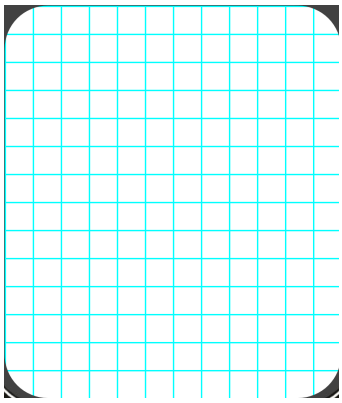
```
Traceback (most recent call last):
  File "<stdin>", line 3
IndentationError: unexpected indent
```

6.3 Zählen mit Schritten, rückwärts zählen

`range` kann eine dritte Zahl bekommen: die **Schrittweite**. `range(0, 240, 20)` liefert 0, 20, 40 … 220. Damit zeichnest du Karopapier:

k06_karopapier.py

```
1 import pbc
2
3 pbc.clear(pbc.WHITE)
4 # Senkrechte Linien: x = 0, 20, 40, ... 220
5 for x in range(0, pbc.WIDTH, 20):
6     pbc.vline(x, 0, pbc.HEIGHT, pbc.CYAN)
7 # Waagrechte Linien: y = 0, 20, 40, ... 260
8 for y in range(0, pbc.HEIGHT, 20):
9     pbc.hline(0, y, pbc.WIDTH, pbc.CYAN)
```

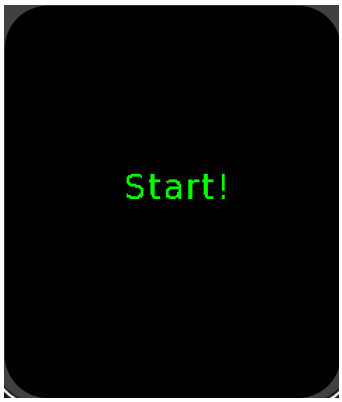


- **Zeile 5–6:** `x` springt in Zwanzigerschritten von 0 bis 220 (240 ist nicht mehr dabei). Für jedes `x` entsteht eine senkrechte Linie über die ganze Höhe.
- **Zeile 8–9:** Dasselbe für die waagrechten Linien, diesmal bis 260.

Rückwärts zählen geht mit einer **negativen** Schrittweite. `range(5, 0, -1)` liefert 5, 4, 3, 2, 1. Die 0 ist wieder nicht dabei. Zusammen mit `time.sleep` wird daraus ein Countdown:

k06_countdown.py

```
1 import pbc
2 import time
3
4 pbc.clear()
5 for zahl in range(5, 0, -1):
6     print(zahl)
7     # Alte Zahl mit einem schwarzen Rechteck uebermalen
8     pbc.fill_rect(100, 110, 50, 30, pbc.BLACK)
9     pbc.text24(f"{zahl}", 112, 115, pbc.WHITE)
10    time.sleep(1)
11 pbc.fill_rect(100, 110, 50, 30, pbc.BLACK)
12 pbc.text24("Start!", 85, 115, pbc.GREEN)
13 print("Start!")
```



In Zeile 8 übermalt ein schwarzes Rechteck die alte Zahl, sonst stünden alle Zahlen übereinander. Zeile 9 macht mit einem f-String (Kapitel 4) aus der Zahl einen Text, Zeile 10 wartet eine Sekunde. Nach der Schleife verschwindet die letzte Zahl, und „Start!“ erscheint.

6.4 Verschachtelte Schleifen

Der Körper einer Schleife darf selbst wieder eine Schleife enthalten (**verschachtelte** Schleifen):

k06_einmaleins.py

```
1 for a in range(1, 3):
2     for b in range(1, 4):
3         print(f"{a} x {b} = {a * b}")
4     print("---")
```

Terminal

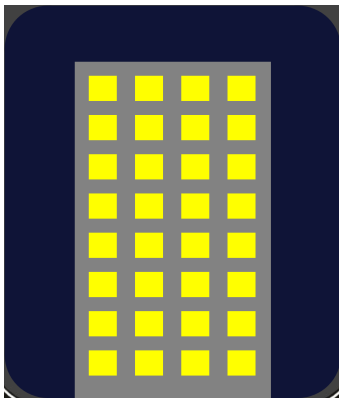
```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
---
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
---
```

Die **äußere** Schleife (Zeile 1) läuft zweimal. Bei **jedem** ihrer Durchläufe läuft die **innere** Schleife (Zeile 2–3) vollständig, also dreimal. Insgesamt wird Zeile 3 also $2 \cdot 3 = 6$-mal ausgeführt. Zeile 4 gehört zur äußeren Schleife (einmal eingerückt) und läuft nach jeder Dreiergruppe.

Verschachtelte Schleifen sind ideal für alles, was in **Zeilen und Spalten** angeordnet ist:

k06_hochhaus.py

```
1 import pbc
2
3 pbc.clear(pbc.rgb(20, 20, 60))
4 pbc.fill_rect(50, 40, 140, 240, pbc.GRAY)
5 for etage in range(8):
6     for fenster in range(4):
7         x = 60 + fenster * 33
8         y = 50 + etage * 28
9         pbc.fill_rect(x, y, 20, 18, pbc.YELLOW)
```



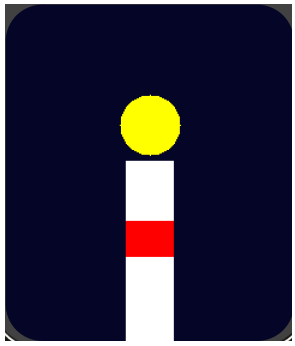
Die äußere Schleife (Zeile 5) geht die 8 Etagen durch, die innere (Zeile 6) in jeder Etage die 4 Fenster. `x` hängt nur vom Fenster ab, `y` nur von der Etage. Zeile 9 läuft $8 \cdot 4 = 32$-mal und zeichnet 32 Fenster.

6.5 `while True` – die Endlosschleife

Eine `for`-Schleife endet, wenn `range` keine Zahlen mehr liefert. Manchmal soll ein Programm aber **immer** weiterlaufen, etwa ein Spiel, eine Uhr oder ein blinkendes Licht. Dafür gibt es `while True:`:

k06_leuchtturm.py

```
 1 import pbc
 2 import time
 3
 4 nacht = pbc.rgb(10, 10, 40)
 5 pbc.clear(nacht)
 6 pbc.fill_rect(100, 130, 40, 150, pbc.WHITE)
 7 pbc.fill_rect(100, 180, 40, 30, pbc.RED)
 8 while True:
 9     pbc.ellipse(120, 100, 25, 25, pbc.YELLOW, fill=True)
10     time.sleep_ms(700)
11     pbc.ellipse(120, 100, 25, 25, nacht, fill=True)
12     time.sleep_ms(300)
13 print("Diese Zeile wird nie erreicht.")
```



- **Zeile 4–7:** Vor der Schleife wird einmal der Turm gezeichnet.
- **Zeile 8:** `while True:` heißt „solange wahr“. `True` ist immer wahr, also wird der eingerückte Körper **ohne Ende** wiederholt.
- **Zeile 9–12:** Das Licht geht an, bleibt 0,7 Sekunden an, wird mit der Hintergrundfarbe übermalt und bleibt 0,3 Sekunden aus.
- **Zeile 13:** Diese Zeile wird **nie** ausgeführt, weil die Schleife davor nicht endet.

Ein Programm mit `while True` hältst du mit ■ **Stop** an. Wie man eine `while`-Schleife mit einer Bedingung beendet, lernst du in Kapitel 7.

Eine Endlosschleife sollte immer eine Pause enthalten, z. B. `time.sleep_ms(20)`. Sonst geht bei Animationen alles so schnell, dass man nichts erkennt.

6.6 Animation – und warum es flackert

Ein Film besteht aus vielen Einzelbildern, die schnell hintereinander gezeigt werden. So machst du es auch auf dem PicoBoy: zeichnen, kurz warten, die Figur ein Stück weiter zeichnen. Vorher muss das alte Bild aber weg, sonst bleiben alle alten Bälle stehen.

Erster Versuch: alles löschen. Wir löschen vor jedem Bild den ganzen Bildschirm:

k06_flackern.py

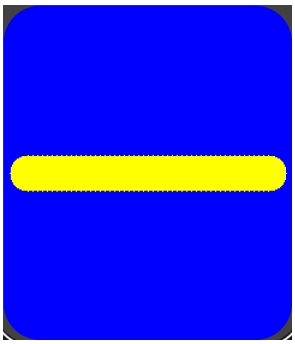
```
1 import pbc
2 import time
3
4 for x in range(20, 221, 4):
5     pbc.clear(pbc.BLUE)
6     pbc.ellipse(x, 140, 15, 15, pbc.YELLOW, fill=True)
7     time.sleep_ms(20)
```

Teste das Programm. Der Ball bewegt sich, aber er **flackert** stark. Der Grund: Jeder Zeichenbefehl erscheint **sofort** auf dem Bildschirm. Zeile 5 füllt den ganzen Bildschirm blau, und das dauert eine Weile, weil dafür alle 67 200 Pixel neu zum Display übertragen werden müssen (etwa 40 ms). In dieser Zeit ist der Ball nicht zu sehen. Erst Zeile 6 zeichnet ihn wieder. Das Display zeigt also abwechselnd „mit Ball“ und „ohne Ball“, und genau das siehst du als Flackern.

Besser: nur die alte Stelle übermalen. Statt des ganzen Bildschirms übermalen wir nur den Ball selbst mit der Hintergrundfarbe:

k06_uebermalen.py

```
1 import pbc
2 import time
3
4 pbc.clear(pbc.BLUE)
5 for x in range(20, 221, 4):
6     pbc.ellipse(x, 140, 15, 15, pbc.YELLOW, fill=True)
7     time.sleep_ms(20)
8     # Alte Stelle blau uebermalen
9     pbc.ellipse(x, 140, 15, 15, pbc.BLUE, fill=True)
10 pbc.ellipse(220, 140, 15, 15, pbc.YELLOW, fill=True)
```



Ohne Zeile 9 zieht der Ball eine Spur.

- **Zeile 4:** Der Hintergrund wird nur **einmal** vor der Schleife gemalt.
- **Zeile 6–7:** Ball zeichnen, kurz warten. In dieser Zeit sieht man ihn.
- **Zeile 9:** Den Ball an derselben Stelle in Blau noch einmal zeichnen. Damit ist er weg, und der Hintergrund ist wieder wie vorher. Gleich danach zeichnet Zeile 6 ihn 4 Pixel weiter rechts.
- **Zeile 10:** Nach der Schleife bleibt der Ball an seiner letzten Stelle stehen.

Jetzt wird nur noch eine kleine Fläche neu gemalt, und das Flackern ist viel schwächer. Diese Abhilfe hat Grenzen: Liegt hinter dem Ball ein Bild statt einer einfarbigen Fläche, übermalt man es mit. Wie man ganz ohne Flackern zeichnet, indem man ein Bild erst im Speicher fertig malt und dann auf einmal anzeigt, lernst du in **Kapitel 17** mit dem Canvas.

Endlos weiterlaufen. Mit `while True` und dem Rest beim Teilen `%` (Kapitel 4) läuft der Ball immer wieder durch:

k06_endlos.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear(pbc.BLUE)
 5 x = 0
 6 while True:
 7     pbc.ellipse(x, 140, 15, 15, pbc.YELLOW, fill=True)
 8     time.sleep_ms(20)
 9     pbc.ellipse(x, 140, 15, 15, pbc.BLUE, fill=True)
10     # Rechts hinaus, links wieder herein: 236, 0, 4, ...
11     x = (x + 4) % pbc.WIDTH
```

Zeile 11 erhöht `x` um 4. Das `% pbc.WIDTH` sorgt dafür, dass `x` nie 240 erreicht: Auf 236 folgt `240 % 240` = 0, der Ball springt an den linken Rand und läuft von vorn. Am Rand ist er dabei halb zu sehen, weil sein Mittelpunkt auf dem Rand liegt. Ein Ball, der am Rand **abprallt**, braucht eine Entscheidung („Ist der Ball am Rand?“). Die lernst du in Kapitel 7.

Ausprobieren: Ein Ladebalken

Ein Balken, der sich beim Start eines Spiels langsam füllt:

k06_ladebalken.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text18("Lade Spiel ...", 65, 90, pbc.WHITE)
 6 # Rahmen mit etwas Luft um den vollen Balken
 7 pbc.rect(18, 128, 204, 24, pbc.WHITE)
 8 for prozent in range(0, 101, 2):
 9     # Farbe wandert von Rot nach Gruen
10     farbe = pbc.rgb(250 - prozent * 2, 50 + prozent * 2, 0)
11     # 2 Pixel Balken pro Prozent
12     pbc.fill_rect(20, 130, prozent * 2, 20, farbe)
13     # Alte Zahl uebermalen, neue schreiben
14     pbc.fill_rect(90, 165, 70, 22, pbc.BLACK)
15     pbc.text18(f"{prozent} %", 100, 165, pbc.WHITE)
16     time.sleep_ms(40)
17 pbc.text24("Fertig!", 82, 210, pbc.YELLOW)
```

- **Zeile 7–8:** Der Rahmen ist etwas größer als der volle Balken (200 Pixel). `prozent` zählt in Zweierschritten bis 100; dafür steht als Ende 101.
- **Zeile 10:** Die Farbe hängt von `prozent` an: Bei 0 ist sie `rgb(250, 50, 0)` (rot), bei 100 `rgb(50, 250, 0)` (grün). Dazwischen wird der Rotanteil kleiner und der Grünanteil größer.
- **Zeile 12:** Der Balken (2 Pixel pro Prozent) wird nur länger, nie kürzer. Deshalb muss nichts gelöscht werden, und er flackert nicht.
- **Zeile 14–15:** Die alte Zahl wird übermalt, dann die neue geschrieben.



Zum Verändern: Ändere das Tempo. Zeichne den Balken senkrecht von unten nach oben wie ein Thermometer.

Übungsaufgaben

Aufgabe 1 ⭐ Was gibt das Programm aus? Schreibe die Ausgabe auf, ohne das Programm zu testen. Prüfe danach im Emulator. Achte genau auf die Einrückung.

k06_aufgabe1.py

```
1 for i in range(2, 12, 3):
2     print(i)
3 print("---")
4 for k in range(3, 0, -1):
5     print("k ist", k)
6 print("doppelt", 2 * k)
```

Aufgabe 2 ⭐ Finde die Fehler Das Programm soll sechs rote Punkte nebeneinander zeichnen, den ersten bei x = 20, den letzten bei x = 220. Es enthält zwei Fehler. Einer meldet sich mit einer Fehlermeldung, der andere zeigt sich erst im Bild.

k06_aufgabe2.py

```
1 import pbc
2
3 pbc.clear()
4 # Sechs rote Punkte nebeneinander, von x = 20 bis x = 220
5 for i in range(1, 6)
6     x = 20 + i * 40
7     pbc.ellipse(x, 140, 10, 10, pbc.RED, fill=True)
```

Aufgabe 3 ⭐⭐ Ein Lineal Zeichne ein Lineal: eine hellbraune Fläche quer über den Bildschirm, alle 10 Pixel ein kurzer Strich und alle 50 Pixel ein langer Strich mit einer Zahl darunter (0, 1, 2 …). Tipp: Nimm zwei Schleifen nacheinander.

Aufgabe 4 ⭐⭐⭐ Abendhimmel Fülle den Bildschirm mit einem Farbverlauf: oben dunkelblau, unten orange. Zeichne dazu für **jede** Pixelzeile `y` eine waagrechte Linie, deren Farbe du aus `y` berechnest. Wie oft läuft deine Schleife?

Aufgabe 5 ⭐⭐⭐ Laufschrift Lass einen Text wie „Heute Sportfest!“ endlos von links nach rechts über den Bildschirm laufen. Tipp: Einen Text löschst du, indem du ihn an derselben Stelle in der Hintergrundfarbe noch einmal schreibst.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `import time`; `time.sleep(s)` wartet s Sekunden (auch `0.5`), `time.sleep_ms(ms)` wartet ms Millisekunden.
- `for i in range(n):` wiederholt den eingerückten Körper n-mal, `i` ist 0, 1 … n − 1. Doppelpunkt und Einrückung (4 Leerzeichen) nicht vergessen.
- `range(start, ende, schritt)`: `ende` ist nie dabei; ein negativer Schritt zählt rückwärts.
- Verschachtelte Schleifen: Die innere läuft bei jedem Durchlauf der äußeren vollständig. Ideal für Zeilen und Spalten.
- `while True:` wiederholt endlos; anhalten mit ■ **Stop**. Immer eine Pause einbauen.
- Animation: zeichnen – warten – übermalen – weiterrücken. Den ganzen Bildschirm zu löschen flackert; nur die alte Stelle zu übermalen flackert viel weniger. Ganz ohne Flackern: Canvas (Kapitel 17).

KAPITEL 7

Entscheidungen

Worum geht's? Bisher hat dein Programm jede Zeile ausgeführt, eine nach der anderen. Jetzt lernst du, wie ein Programm **entscheidet**: Es prüft eine Bedingung und macht dann das eine oder das andere. Außerdem lernst du eine Schleife kennen, die so lange läuft, wie eine Bedingung erfüllt ist. Am Ende programmierst du ein Ratespiel.

7.1 `if`, `else`, `elif`

Wenn …, sonst … Darfst du Achterbahn fahren? Das hängt von deiner Größe ab:

k07_achterbahn.py

```
1 groesse = int(input("Größe in cm? "))
2
3 if groesse >= 140:
4     print("Du darfst mitfahren!")
5 else:
6     print("Leider zu klein.")
7     print("Komm nächstes Jahr wieder.")
8
9 print("Der Nächste, bitte!")
```

Terminal

```
Größe in cm? 128
Leider zu klein.
Komm nächstes Jahr wieder.
Der Nächste, bitte!
```

- Zeile 1 kennst du aus Kapitel 3: `input` liest einen Text, `int` macht eine ganze Zahl daraus.
- Zeile 3 enthält eine **Bedingung**: `if groesse >= 140:` heißt „wenn die Größe mindestens 140 ist". Am Ende der Zeile steht ein **Doppelpunkt**.
- Zeile 4 ist **eingerückt**, sie beginnt mit vier Leerzeichen. Sie gehört zum `if` und läuft nur, wenn die Bedingung erfüllt ist. Eingerückte Zeilen, die zusammengehören, nennt man einen **Block**.
- `else:` in Zeile 5 heißt „sonst". Der Block darunter (Zeilen 6 und 7) läuft nur, wenn die Bedingung **nicht** erfüllt ist. Es läuft also immer genau einer der beiden Blöcke. `else` steht genau so weit links wie sein `if`.
- Zeile 9 ist nicht eingerückt. Sie gehört nicht mehr dazu und läuft immer.

Das `else` mit seinem Block darf auch fehlen. Dann passiert im anderen Fall einfach nichts:

```
if temperatur > 27:
    print("Hitzefrei!")
```

Mehr als zwei Fälle: `elif`. Mit `elif` (kurz für „else if“, „sonst wenn“) prüfst du nacheinander mehrere Bedingungen:

k07_kleidung.py

```
temperatur = int(input("Grad draußen? "))

if temperatur >= 25:
    print("T-Shirt und kurze Hose")
elif temperatur >= 15:
    print("T-Shirt und Pulli")
elif temperatur >= 5:
    print("Jacke")
else:
    print("Mütze, Schal und Jacke")
```

Terminal

```
Grad draußen? 12
Jacke
```

Python prüft die Bedingungen **von oben nach unten**. Bei 12 °C ist `temperatur >= 25` falsch, `temperatur >= 15` auch, aber `temperatur >= 5` stimmt: „Jacke“. Sobald eine Bedingung zutrifft, läuft ihr Block, und alle weiteren `elif` und das `else` werden übersprungen. Deshalb ist die Reihenfolge wichtig: Stünde `temperatur >= 5` ganz oben, bekäme man auch bei 30 °C eine Jacke empfohlen. Das `else` am Ende ist für alle übrigen Fälle da. Es darf auch fehlen.

Doppelpunkt vergessen. Hinter `if`, `elif` und `else` muss ein Doppelpunkt stehen. Fehlt er, kann Python das Programm nicht lesen:

k07_fehler_doppelpunkt.py

```
punkte = 120

if punkte > 100
    print("Neuer Rekord!")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 3
SyntaxError: invalid syntax
```

Das Hinweisfenster von PBC+ Studio nennt den fehlenden Doppelpunkt gleich als erste mögliche Ursache.

7.2 Vergleiche: `==`, `!=`, `<`, `<=`, `>`, `>=`

Eine Bedingung ist meistens ein **Vergleich**. Python kennt sechs Vergleichsoperatoren:

Operator	Bedeutung	Beispiel	Ergebnis
`==`	gleich	`3 == 3`	`True`
`!=`	ungleich	`3 != 3`	`False`
`<`	kleiner als	`2 < 5`	`True`
`<=`	kleiner oder gleich	`5 <= 5`	`True`
`>`	größer als	`2 > 5`	`False`
`>=`	größer oder gleich	`7 >= 5`	`True`

Ein Vergleich liefert einen von zwei Werten: `True` (wahr) oder `False` (falsch). Mit `print` kannst du dir das Ergebnis ansehen:

k07_vergleiche.py

```
1 a = 7
2 b = 12
3
4 print(a < b)
5 print(a > b)
6 print(a == 7)
7 print(a != 7)
8 print(b >= 12)
9 print(a + 5 == b)
10 print("Lena" == "lena")
```

Terminal

```
True
False
True
False
True
True
False
```

Zeile 9 zeigt: Auf beiden Seiten darf auch eine Rechnung stehen. Python rechnet zuerst `a + 5` aus und vergleicht dann. Zeile 10 zeigt: Auch Texte lassen sich vergleichen, und dabei zählt jeder Buchstabe genau, auch Groß- und Kleinschreibung.

`=` und `==` verwechseln. Ein einfaches `=` ist eine **Zuweisung**: Es legt einen Wert in einer Variablen ab. Ein doppeltes `==` **vergleicht**. Schreibst du in einer Bedingung `=`, meldet Python einen Fehler:

k07_fehler_gleich.py

```
1 wurf = 6
2
3 if wurf = 6:
4     print("Sechs! Du darfst noch einmal.")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 3
SyntaxError: invalid syntax
```

Text oder Zahl? Dieses Programm sieht richtig aus, sagt aber immer „Falsch!“:

k07_text_zahl.py

```
1 antwort = input("Was ist 3 mal 4? ")
2
3 if antwort == 12:
4     print("Richtig!")
5 else:
6     print("Falsch!")
```

Terminal

```
Was ist 3 mal 4? 12
Falsch!
```

`input` liefert immer einen **Text**, hier den Text `"12"`. Der Text `"12"` ist aber nicht dasselbe wie die Zahl `12`. Der Vergleich in Zeile 3 ist deshalb immer `False`, und es gibt nicht einmal eine Fehlermeldung. Abhilfe: `antwort = int(input("Was ist 3 mal 4? "))`.

> Bei `<` und `>` merkt Python den Unterschied dagegen sofort: Vergleichst du einen Text mit einer Zahl, bricht das Programm mit `TypeError: unsupported types for __lt__: 'str', 'int'` ab (`__lt__` steht für „less than“, kleiner als). Auch dann fehlt meistens das `int()` um das `input`.

7.3 Wahr und falsch: `True`, `False`, `and`, `or`, `not`

`True` und `False` sind Werte wie Zahlen und Texte. Du kannst sie in Variablen speichern. Ihr Datentyp heißt `bool` (nach dem Mathematiker George Boole), man spricht auch von **Wahrheitswerten**. Mit drei Wörtern kannst du Wahrheitswerte verknüpfen:

Ausdruck	ist `True`, wenn …
`a and b`	**beide** wahr sind
`a or b`	**mindestens einer** wahr ist (oder beide)
`not a`	`a` falsch ist (`not` dreht um)

Ein Beispiel: Ist `regen = True` und `sonne = False`, dann ist `regen and sonne` falsch, `regen or sonne` wahr und `not regen` falsch. Auch das Ergebnis eines Vergleichs kannst du speichern: Nach `warm = temperatur > 20` steht in `warm` entweder `True` oder `False`.

Mit `and` und `or` kannst du mehrere Bedingungen zu einer verbinden. Ein Kinofilm „ab 12“ darf von Kindern ab 6 Jahren besucht werden, wenn die Eltern dabei sind:

k07_kino.py

```
1 alter = int(input("Wie alt bist du? "))
2 antwort = input("Mit Eltern? (j/n) ")
3 mit_eltern = antwort == "j"
4
5 # Film ab 12: ab 12 allein, ab 6 mit den Eltern
6 if alter >= 12 or (alter >= 6 and mit_eltern):
7     print("Viel Spaß im Film!")
8 else:
9     print("Der Film ist nichts für dich.")
```

Terminal

```
Wie alt bist du? 9
Mit Eltern? (j/n) j
Viel Spaß im Film!
```

- In Zeile 3 wird die Antwort mit `"j"` verglichen. Das Ergebnis, `True` oder `False`, kommt in die Variable `mit_eltern`.
- Zeile 6 liest sich wie ein Satz: „Wenn das Alter mindestens 12 ist **oder** (das Alter mindestens 6 ist **und** die Eltern dabei sind)“. Die Klammern zeigen, was zusammengehört. Wie beim Rechnen wird zuerst die Klammer ausgewertet.
- `if mit_eltern:` ist bereits eine vollständige Bedingung. Du musst nicht `if mit_eltern == True:` schreiben.

> Für „liegt zwischen“ kannst du Vergleiche wie in der Mathematik hintereinanderschreiben: `if 1 <= wurf <= 6:` bedeutet dasselbe wie `if wurf >= 1 and wurf <= 6:`.

7.4 Einrückung: Was gehört wozu?

In Python entscheidet allein die **Einrückung**, welche Zeilen zu einem `if` gehören. Die beiden folgenden Programme unterscheiden sich nur in Zeile 5:

k07_einrueckung1.py

```
1 punkte = int(input("Punkte? "))
2
3 if punkte >= 100:
4     print("Neuer Rekord!")
5     print("Glückwunsch!")
6 print("Spiel vorbei.")
```

Terminal

```
Punkte? 40
Spiel vorbei.
```

k07_einrueckung2.py

```
1 punkte = int(input("Punkte? "))
2
3 if punkte >= 100:
4     print("Neuer Rekord!")
5 print("Glückwunsch!")
6 print("Spiel vorbei.")
```

Terminal

```
Punkte? 40
Glückwunsch!
Spiel vorbei.
```

Links gehört „Glückwunsch!“ zum `if`. Rechts steht die Zeile ganz links und wird **immer** ausgeführt, auch bei 40 Punkten. Python meldet hier keinen Fehler, denn beide Programme sind erlaubt. Nur du weißt, welches du gemeint hast.

Die Regeln:

- Nach jeder Zeile, die mit einem Doppelpunkt endet (`if`, `elif`, `else`, `for`, `while`), folgt ein eingerückter Block mit mindestens einer Zeile.
- Alle Zeilen eines Blocks sind **gleich weit** eingerückt. Üblich sind 4 Leerzeichen. Der Editor rückt nach einem Doppelpunkt automatisch ein.
- Der Block endet mit der ersten Zeile, die wieder weiter links steht.

Stimmt die Einrückung nicht, bricht Python ab, bevor das Programm startet:

k07_fehler_einrueckung.py

```
1 punkte = 120
2
3 if punkte >= 100:
4 print("Neuer Rekord!")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4
SyntaxError: invalid syntax
```

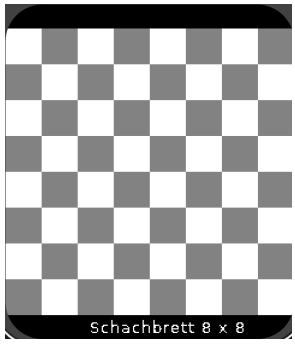
Hier fehlt die Einrückung nach dem `if`. MicroPython meldet dafür einen `SyntaxError` in der Zeile, die eingerückt sein müsste. Ist umgekehrt eine Zeile weiter eingerückt als die Zeile darüber, ohne dass ein neuer Block beginnt, heißt die Meldung `IndentationError: unexpected indent` („unerwartete Einrückung“).

Blöcke in Blöcken. Ein `if` kann in einer Schleife stehen, und eine Schleife in einer anderen. Jede Ebene rückt um weitere vier Leerzeichen ein:

k07_schachbrett.py

```
 1 import pbc
 2
 3 pbc.clear()
 4 for zeile in range(8):
 5     for spalte in range(8):
 6         x = spalte * 30
 7         y = 20 + zeile * 30
 8         # Felder, bei denen zeile + spalte gerade ist, sind hell
 9         if (zeile + spalte) % 2 == 0:
10             farbe = pbc.WHITE
11         else:
12             farbe = pbc.GRAY
13         pbc.fill_rect(x, y, 30, 30, farbe)
14 pbc.text12("Schachbrett 8 x 8", 70, 262, pbc.WHITE)
```

- Die beiden `for`-Schleifen (Zeilen 4 und 5) laufen über 8 Zeilen und 8 Spalten, wie in Kapitel 6.
- Das `if` in Zeile 9 steht in der inneren Schleife. Es entscheidet für **jedes** der 64 Felder neu, welche Farbe es bekommt: Ist `zeile + spalte` gerade (Rest 0 beim Teilen durch 2), wird das Feld weiß, sonst grau.
- Zeile 13 gehört wieder zur inneren Schleife, aber nicht mehr zum `if` und nicht zum `else`. Sie läuft also für jedes Feld.
- Zeile 14 steht ganz links. Sie läuft erst, wenn beide Schleifen fertig sind.



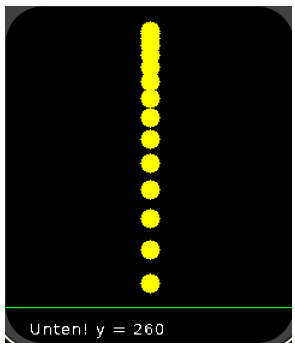
7.5 Die `while`-Schleife mit Bedingung, `break`

Aus Kapitel 6 kennst du `while True:`, die Endlosschleife. Statt `True` kann hinter `while` jede Bedingung stehen. Dann wiederholt Python den Block, **solange** die Bedingung erfüllt ist. Wie oft das ist, muss vorher niemand wissen. Ein Ball fällt und wird dabei immer schneller. Wann ist er unten? Damit man die Bewegung sieht, wird der alte Ball nicht gelöscht:

k07_ball.py

```
 1 import pbc
 2 import time
 3
 4 y = 20
 5 tempo = 0
 6
 7 pbc.clear()
 8 # Solange der Ball noch nicht unten ist: zeichnen, schneller werden
 9 while y < 240:
10     pbc.ellipse(120, y, 8, 8, pbc.YELLOW, fill=True)
11     time.sleep_ms(100)
12     tempo += 2
13     y += tempo
14
15 pbc.hline(0, 250, 240, pbc.GREEN)
16 pbc.text12(f"Unten! y = {y}", 20, 260, pbc.WHITE)
```

- Zeile 9 prüft: Ist der Ball noch über y = 240?
- In Zeile 12 wird `tempo` größer, in Zeile 13 rückt der Ball um `tempo` Pixel weiter. Deshalb werden die Abstände zwischen den Bällen immer größer.
- Beim letzten Durchlauf springt `y` von 230 auf 260. Danach ist `y < 240` falsch, die Schleife endet, und der Ball bei y = 260 wird gar nicht mehr gezeichnet. Das zeigt die Ausgabe in Zeile 16.



> **`for` oder `while`?** Weißt du vorher, wie oft etwas wiederholt wird (10 Sterne zeichnen), nimm `for`. Hängt es von etwas ab, das sich erst während des Laufs ergibt (bis der Ball unten ist, bis die Antwort stimmt), nimm `while`.

Warten auf die richtige Eingabe. Eine `while`-Schleife kann so lange fragen, bis die Antwort stimmt:

k07_passwort.py

```
1 passwort = input("Passwort: ")
2
3 while passwort != "Sesam":
4     print("Falsch! Noch einmal.")
5     passwort = input("Passwort: ")
6
7 print("Die Tür öffnet sich.")
```

Terminal

```
Passwort: sesam
Falsch! Noch einmal.
Passwort: Sesam öffne dich
Falsch! Noch einmal.
Passwort: Sesam
Die Tür öffnet sich.
```

Beachte Zeile 5: In der Schleife muss die Variable `passwort` neu belegt werden. Fehlt diese Zeile, ändert sich die Bedingung nie, und die Schleife schreibt endlos „Falsch!“ ins Terminal. Dann hilft nur ■ **Stop**.

Aussteigen mit `break`. Manchmal merkt man erst mitten im Block, dass die Schleife enden soll. Dann beendet `break` die Schleife sofort, und das Programm macht mit der ersten Zeile **nach** der Schleife weiter:

k07_summe.py

```
1 summe = 0
2
3 while True:
4     zahl = int(input("Zahl (0 = Ende): "))
5     if zahl == 0:
6         break
7     summe += zahl
8
9 print(f"Summe: {summe}")
```

Terminal

```
Zahl (0 = Ende): 12
Zahl (0 = Ende): 7
Zahl (0 = Ende): 23
Zahl (0 = Ende): 0
Summe: 42
```

`while True:` würde ewig laufen. Erst die Eingabe 0 führt in Zeile 6 zum `break`, und Zeile 9 gibt die Summe aus. Die 0 selbst wird nicht mehr addiert, weil Zeile 7 nach dem `break` nicht mehr drankommt. `break` funktioniert genauso in einer `for`-Schleife.

Ausprobieren: Zahlen raten

Der PicoBoy denkt sich eine Zahl von 1 bis 100. Du hast sieben Versuche, sie zu erraten. Nach jedem Tipp verrät das Programm, ob die Zahl größer oder kleiner ist.

k07_zahlenraten.py

```
 1 import random
 2
 3 geheim = random.randint(1, 100)
 4 versuche = 0
 5 print("Ich denke mir eine Zahl von 1 bis 100.")
 6 print("Du hast 7 Versuche.")
 7
 8 while versuche < 7:
 9     tipp = int(input("Dein Tipp: "))
10     versuche += 1
11     if tipp < geheim:
12         print("Zu klein!")
13     elif tipp > geheim:
14         print("Zu groß!")
15     else:
16         print(f"Richtig! Du hast {versuche} Versuche gebraucht.")
17         break
18
19 if tipp != geheim:
20     print(f"Schade! Die Zahl war {geheim}.")
```

- Zeile 3 wählt die Geheimzahl mit `random.randint` (Kapitel 4).
- Die Schleife in Zeile 8 läuft, solange noch Versuche übrig sind. Nach dem siebten Tipp ist `versuche < 7` falsch.
- Die Zeilen 11 bis 17 entscheiden mit `if`, `elif` und `else`: zu klein, zu groß oder richtig. Bei einem Treffer beendet `break` die Schleife sofort.
- Nach der Schleife ist nicht klar, *warum* sie zu Ende ist: wegen eines Treffers oder weil die Versuche aufgebraucht sind. Zeile 19 prüft das. Nur wenn der letzte Tipp falsch war, wird die Lösung verraten.

Testen mit einer festen Zahl. Bei einer Zufallszahl weißt du nicht, was das Programm antworten müsste. Ersetze deshalb zum Testen Zeile 3 durch `geheim = 34`. Jetzt kannst du jede Antwort überprüfen:

Terminal (Testfassung mit geheim = 34)

```
Ich denke mir eine Zahl von 1 bis 100.
Du hast 7 Versuche.
Dein Tipp: 50
Zu groß!
Dein Tipp: 25
Zu klein!
Dein Tipp: 37
Zu groß!
Dein Tipp: 31
Zu klein!
Dein Tipp: 34
Richtig! Du hast 5 Versuche gebraucht.
```

Ist alles richtig, kommt die Zufallszahl wieder hinein.

> Die beste Strategie: Rate immer die Mitte des Bereichs, der noch möglich ist. So halbierst du den Bereich mit jedem Tipp, und sieben Versuche reichen für jede Zahl von 1 bis 100.

Zum Verändern: Lass den Bereich (1 bis 100) und die Zahl der Versuche am Anfang eingeben. Gib nach einem falschen Tipp zusätzlich „heiß!“ aus, wenn er höchstens 5 von der Geheimzahl entfernt ist. Zeige das Ergebnis am Ende zusätzlich groß auf dem Display.

Übungsaufgaben

Aufgabe 1 ⭐ Was gibt das Programm aus? Lies das Programm, ohne es zu testen. Welche Buchstaben erscheinen im Terminal? Prüfe danach im Emulator.

k07_aufgabe1.py

```
punkte = 75
leben = 2
bonus = False

if punkte > 100:
    print("A")
elif punkte > 50:
    print("B")
else:
    print("C")
if leben > 0 and not bonus:
    print("D")
if punkte < 50 or leben == 2:
    print("E")
if bonus:
    print("F")
print("G")
```

Aufgabe 2 ⭐ Finde die Fehler Das Programm soll sagen, ob man volljährig (mindestens 18 Jahre alt) ist. Es enthält vier Fehler. Drei davon meldet Python, einer ist ein Denkfehler.

k07_aufgabe2.py

```
alter = input("Wie alt bist du? ")

if alter = 18
    print("Du bist volljährig.")
else:
print("Du bist noch nicht volljährig.")
```

Aufgabe 3 ⭐⭐ Die Ergebnis-Ampel Schreibe ein Programm, das nach einer Punktzahl von 0 bis 100 fragt und auf dem Display einen großen Kreis zeigt: grün ab 80 Punkten, gelb ab 50 Punkten, sonst rot. Darunter stehen die Punktzahl und ein passender Text.

Aufgabe 4 ⭐⭐ Schaltjahr Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, aber nicht durch 100. Durch 400 teilbare Jahre sind aber doch Schaltjahre. Schreibe ein Programm, das nach einer Jahreszahl fragt und sagt, ob sie ein Schaltjahr ist. Teste mit 2024, 2023, 1900 und 2000. *Tipp:* `jahr % 4 == 0` prüft, ob `jahr` durch 4 teilbar ist.

Aufgabe 5 ⭐⭐ Papier falten Ein Blatt Papier ist 0,1 mm dick. Faltest du es einmal, ist der Stapel 0,2 mm dick, nach zwei Faltungen 0,4 mm. Wie oft müsste man falten, damit der Stapel höher ist als der Mount Everest (8848 m)? Schreibe ein Programm mit einer `while`-Schleife, das es ausrechnet. *Tipp:* Rechne in Mikrometern (1 mm = 1000 Mikrometer), dann kommst du mit ganzen Zahlen aus.

Aufgabe 6 ⭐⭐⭐ Kopfrechentrainer Das Programm stellt fünf Aufgaben aus dem kleinen Einmaleins mit Zufallszahlen von 2 bis 9, zum Beispiel `7 mal 4 =` . Es sagt nach jeder Antwort, ob sie stimmt, und zählt die richtigen. Wer 0 eingibt, beendet das Training sofort. Am Ende erscheinen das Ergebnis („3 von 5“) und ein Lob, das zum Ergebnis passt.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `if bedingung:` führt den eingerückten Block nur aus, wenn die Bedingung wahr ist. `elif` prüft weitere Bedingungen, `else` fängt den Rest auf. Es läuft höchstens ein Block, der erste passende.
- Vergleiche: `==`, `!=`, `<`, `<=`, `>`, `>=`. Sie liefern `True` oder `False`. Vergleichen mit `==`, zuweisen mit `=`.
- `input` liefert Text. Vor einem Vergleich mit Zahlen `int()` benutzen.
- `and`: beide wahr, `or`: mindestens eins wahr, `not`: umdrehen.
- Die Einrückung (4 Leerzeichen) bestimmt, was zu einem Block gehört. Hinter `if`, `elif`, `else`, `for`, `while` steht ein Doppelpunkt.
- `while bedingung:` wiederholt, solange die Bedingung wahr ist. `break` beendet eine Schleife sofort.

KAPITEL 8

Tasten und Joystick

Worum geht's? Jetzt wird es interaktiv: Dein Programm reagiert auf den Joystick und die Tasten A und B. Du lernst den Unterschied zwischen „Taste ist gerade unten" und „Taste wurde gedrückt", baust eine Spielschleife und misst Zeiten auf die Millisekunde genau. Am Ende steht ein Reaktionstest.

Der PicoBoy hat sieben Eingaben: den Joystick mit den Richtungen hoch, runter, links und rechts, das Drücken des Joysticks (die Taste **MITTE**) und die Tasten **A** und **B**. Im Emulator benutzt du dafür die Tastatur:

Am PicoBoy	Im Emulator	Name in Python
Joystick hoch / runter / links / rechts	Pfeiltasten	`up`, `down`, `left`, `right`
Joystick drücken (MITTE)	Leertaste	`center`
Taste A / Taste B	Taste **A** / Taste **B**	`a`, `b`

> Reagiert der Emulator nicht auf die Tastatur, klicke einmal auf das Bild des PicoBoy im Emulator und drücke die Taste dann noch einmal. Du kannst die Tasten im Bild auch mit der Maus anklicken.

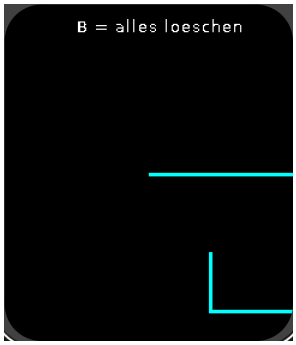
8.1 „wird gedrückt": `pbc.pressed_a()` und Co.

`pbc.pressed_a()` fragt nach, ob die Taste A **in diesem Moment** unten ist. Die Funktion gibt `True` oder `False` zurück. Ein Wert, den eine Funktion zurückgibt, heißt **Rückgabewert**. Weil er ein Wahrheitswert ist, kannst du die Funktion direkt als Bedingung in ein `if` schreiben. Für jede Taste gibt es eine eigene Funktion: `pressed_up()`, `pressed_down()`, `pressed_left()`, `pressed_right()`, `pressed_center()`, `pressed_a()`, `pressed_b()`.

Die Zaubertafel malt mit dem Joystick Linien. Der alte Punkt wird nie gelöscht, so entsteht eine Spur:

k08_zaubertafel.py

```
1  import pbc
2  import time
3
4  x = 120
5  y = 140
6
7  pbc.clear()
8  pbc.text12("B = alles loeschen", 60, 10, pbc.WHITE)
9
10 while True:
11     if pbc.pressed_left():
12         x -= 2
13     if pbc.pressed_right():
14         x += 2
15     if pbc.pressed_up():
16         y -= 2
17     if pbc.pressed_down():
18         y += 2
19     if pbc.pressed_b():
20         pbc.clear()
21     pbc.fill_rect(x, y, 3, 3, pbc.CYAN)
22     time.sleep_ms(10)
```



- `x` und `y` (Zeilen 4 und 5) speichern, wo der Stift gerade ist.
- Die Endlosschleife ab Zeile 10 fragt bei jedem Durchlauf alle Richtungen ab. Solange du zum Beispiel rechts drückst, ist `pbc.pressed_right()` bei jedem Durchlauf `True`, und `x` wächst um 2.
- Es sind vier einzelne `if` und kein `if`/`elif`. So werden **alle** Richtungen geprüft, auch wenn schon eine zutraf. Drückst du rechts und runter zugleich, ändern sich `x` und `y`: Der Stift malt schräg.
- Zeile 21 malt an der neuen Stelle ein Quadrat von 3 × 3 Pixeln.
- Die Pause in Zeile 22 bestimmt die Geschwindigkeit: Je kürzer sie ist, desto öfter läuft die Schleife und desto schneller bewegt sich der Stift.

> **Klammern vergessen.** `if pbc.pressed_a:` (ohne `()`) ist kein Fehler für Python, fragt aber keine Taste ab. Ohne Klammern ist `pbc.pressed_a` die Funktion selbst und nicht ihr Ergebnis, und eine Funktion zählt in einer Bedingung immer als wahr. Der Block läuft also bei jedem Durchlauf.

8.2 „wurde gedrückt“: `pbc.was_pressed_a()` – genau einmal pro Druck

Ein Zähler am Eingang eines Museums soll bei jedem Druck auf A um 1 weiter zählen:

k08_zaehler.py

```
 1 import pbc
 2 import time
 3
 4 besucher = 0
 5 pbc.clear()
 6 pbc.text18("Besucher", 80, 80, pbc.WHITE)
 7 pbc.text24("0", 112, 120, pbc.YELLOW)
 8
 9 while True:
10     if pbc.was_pressed_a():
11         besucher += 1
12         print("Besucher:", besucher)
13         pbc.fill_rect(0, 115, 240, 35, pbc.BLACK)
14         pbc.text24(str(besucher), 112, 120, pbc.YELLOW)
15     time.sleep_ms(20)
```



Zeile 10 benutzt `pbc.was_pressed_a()`. Diese Funktion liefert `True`, wenn A seit der letzten Abfrage **neu gedrückt** wurde, und zwar nur ein einziges Mal pro Druck. Danach ist sie wieder `False`, bis die Taste losgelassen und erneut gedrückt wird. Bei einer Änderung löscht Zeile 13 die alte Zahl mit einem schwarzen Rechteck, Zeile 14 schreibt die neue. `str` macht aus der Zahl einen Text (Kapitel 3).

Was passiert, wenn man in Zeile 10 stattdessen `pbc.pressed_a()` schreibt? Beide Programme wurden mit drei kurzen Tastendrücken bzw. einem Tastendruck von 0,3 Sekunden getestet:

mit was_pressed_a(): 3 Drücke

```
Besucher: 1
Besucher: 2
Besucher: 3
```

mit pressed_a(): 1 Druck

```
Besucher: 1
Besucher: 2
Besucher: 3
Besucher: 4
Besucher: 5
Besucher: 6
Besucher: 7
```

Mit `pressed_a()` zählt jeder Schleifendurchlauf, in dem die Taste unten ist. Selbst ein kurzer Druck dauert mehrere Durchläufe.

`was_pressed` merkt sich den Druck. Der PicoBoy bemerkt einen Druck auch dann, wenn das Programm gerade mit etwas anderem beschäftigt ist. Das zeigt das folgende Programm. Während der drei Sekunden in Zeile 5 wurde A kurz gedrückt und wieder losgelassen:

k08_merken.py

```
1 import pbc
2 import time
3
4 print("Schnell kurz A drücken!")
5 time.sleep(3)
6 print(pbc.pressed_a())
7 print(pbc.was_pressed_a())
8 print(pbc.was_pressed_a())
```

Terminal

```
Schnell kurz A drücken!
False
True
False
```

`pressed_a()` in Zeile 6 meldet `False`, denn jetzt ist die Taste nicht mehr unten. `was_pressed_a()` in Zeile 7 hat sich den Druck gemerkt und meldet `True`. Die zweite Abfrage in Zeile 8 liefert `False`: Der Druck wurde schon „abgeholt".

Du willst …	Nimm …
etwas tun, **solange** die Taste unten ist (bewegen, Dauerfeuer)	`pressed_…()`
etwas **einmal pro Druck** tun (zählen, umschalten, Menü, springen)	`was_pressed_…()`

8.3 Auf eine Taste warten

Oft soll ein Programm warten, bis jemand bereit ist. Dafür gibt es `pbc.wait_for_button()`. Die Funktion wartet, bis **irgendeine** Taste gedrückt und wieder losgelassen wurde:

k08_start.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text18("Weltraumflug", 65, 100, pbc.WHITE)
5 pbc.text12("Druecke eine Taste", 70, 140, pbc.GRAY)
6
7 pbc.wait_for_button()
8
9 pbc.clear(pbc.BLUE)
10 pbc.text24("Los geht's!", 60, 125, pbc.YELLOW)
```

Soll es eine **bestimmte** Taste sein, baust du die Warteschleife selbst:

```
while not pbc.was_pressed_a():
    time.sleep_ms(10)
```





Lies das so: „Solange A **nicht** gedrückt wurde, warte 10 ms." Sobald A gedrückt wird, ist die Bedingung falsch, und das Programm geht weiter. Mit `break` kannst du auf eine von mehreren Tasten warten und dir merken,

welche es war:

k08_quiz.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text18("Ist ein Wal", 70, 80, pbc.WHITE)
 6 pbc.text18("ein Fisch?", 75, 105, pbc.WHITE)
 7 pbc.text12("A = Ja      B = Nein", 60, 140, pbc.GRAY)
 8
 9 # Warten, bis A oder B gedrueckt wird
10 while True:
11     if pbc.was_pressed_a():
12         antwort = "Ja"
13         break
14     if pbc.was_pressed_b():
15         antwort = "Nein"
16         break
17     time.sleep_ms(10)
18
19 if antwort == "Nein":
20     pbc.clear(pbc.GREEN)
21     pbc.text24("Richtig!", 75, 100, pbc.WHITE)
22     pbc.text12("Wale sind Saeugetiere.", 55, 150, pbc.WHITE)
23 else:
24     pbc.clear(pbc.RED)
25     pbc.text24("Leider falsch", 45, 120, pbc.WHITE)
```

- Die Schleife in Zeile 10 läuft endlos, bis eine der beiden Tasten gedrückt wurde. Dann speichert sie die Antwort in `antwort` und steigt mit `break` aus.
- Ab Zeile 19 wird die Antwort ausgewertet. Nach einem Druck auf B wird der Bildschirm grün und zeigt „Richtig!“.



8.4 Die Spielschleife: lesen – rechnen – zeichnen – warten

Fast jedes Spiel ist eine Endlosschleife, die immer wieder dieselben vier Schritte macht. Ein Durchlauf heißt **Frame** (Einzelbild):

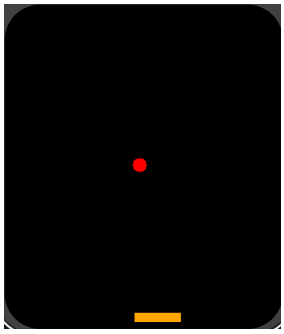
1. **Lesen:** Welche Tasten sind gedrückt?
2. **Rechnen:** Wohin bewegen sich die Figuren? Gibt es einen Treffer? Ändert sich der Punktestand?
3. **Zeichnen:** Den neuen Zustand auf den Bildschirm bringen.
4. **Warten:** Eine kurze Pause. Sie bestimmt, wie schnell das Spiel läuft.

Im folgenden Spiel fällt ein Apfel, und du fängst ihn mit einem Korb, den du nach links und rechts bewegst. Die vier Schritte sind durch Kommentare markiert:

k08_korb.py

```
 1 import pbc
 2 import time
 3 import random
 4
 5 korb_x = 100                    # linke Kante des Korbs
 6 apfel_x = random.randint(10, 230)
 7 apfel_y = 0
 8 punkte = 0
 9 pbc.clear()
10
11 while True:
12     # 1. Lesen: Joystick abfragen
13     links = pbc.pressed_left()
14     rechts = pbc.pressed_right()
15
16     # 2. Rechnen: alte Stellen merken, neue berechnen
17     alt_korb_x = korb_x
18     alt_apfel_x = apfel_x
19     alt_apfel_y = apfel_y
20     if links and korb_x > 0:
21         korb_x -= 4
22     if rechts and korb_x < 200:
23         korb_x += 4
24     apfel_y += 3
25     if apfel_y > 262:           # Apfel ist unten angekommen
26         if apfel_x >= korb_x and apfel_x <= korb_x + 40:
27             punkte += 1
28             print("Gefangen! Punkte:", punkte)
29         apfel_x = random.randint(10, 230)
30         apfel_y = 0
31
32     # 3. Zeichnen: an der alten Stelle loeschen, an der neuen malen
33     pbc.fill_rect(alt_korb_x, 266, 40, 8, pbc.BLACK)
34     pbc.ellipse(alt_apfel_x, alt_apfel_y, 6, 6, pbc.BLACK, fill=True)
35     pbc.fill_rect(korb_x, 266, 40, 8, pbc.ORANGE)
36     pbc.ellipse(apfel_x, apfel_y, 6, 6, pbc.RED, fill=True)
37
38     # 4. Warten
39     time.sleep_ms(20)
```

- **Lesen** (Zeilen 13 und 14): Die Ergebnisse von `pressed_left()` und `pressed_right()` kommen in Variablen. So wird jede Taste nur einmal pro Frame abgefragt.
- **Rechnen** (Zeilen 17 bis 30): Zuerst merkt sich das Programm die alten Positionen, sie werden gleich zum Löschen gebraucht. Mit `and` wird der Korb nur bewegt, wenn er noch nicht am Rand ist (Zeilen 20 und 22).



Der Korb ist 40 Pixel breit, also darf seine linke Kante höchstens bei 200 liegen. Ist der Apfel unten (Zeile 25), prüft Zeile 26, ob er über dem Korb ist. Dann startet ein neuer Apfel oben.

- **Zeichnen** (Zeilen 33 bis 36): Korb und Apfel werden an der alten Stelle schwarz übermalt und an der neuen gezeichnet. Das flackert viel weniger, als jedes Mal den ganzen Bildschirm zu löschen (Kapitel 6).
- **Warten** (Zeile 39): 20 ms Pause.

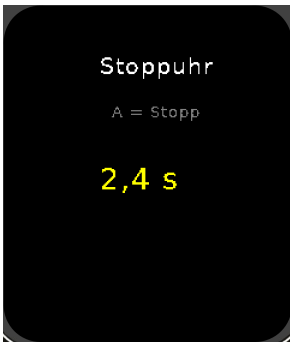
> Halte die Schritte auch in deinen eigenen Spielen getrennt. Wenn etwas nicht stimmt, weißt du dann sofort, wo du suchen musst: Bewegt sich die Figur falsch, liegt es am Rechnen; sieht sie falsch aus, am Zeichnen.

8.5 Zeit messen mit `time.ticks_ms()`

`time.ticks_ms()` liefert einen Zähler, der jede Millisekunde um 1 wächst. Auf dem Gerät zählt er seit dem Einschalten, im Emulator ungefähr seit dem Start des Emulators. Der Wert selbst ist also nicht interessant, wohl aber der **Unterschied** zwischen zwei Zeitpunkten. Den berechnet `time.ticks_diff(spaeter, frueher)`. Achte auf die Reihenfolge: zuerst der spätere Zeitpunkt, sonst kommt eine negative Zahl heraus.

k08_stoppuhr.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text18("Stoppuhr", 80, 40, pbc.WHITE)
 6 pbc.text12("A = Start", 90, 80, pbc.GRAY)
 7 while not pbc.was_pressed_a():
 8     time.sleep_ms(10)
 9
10 start = time.ticks_ms()
11 pbc.text12("A = Stopp", 90, 80, pbc.GRAY, bg=pbc.BLACK)
12 while not pbc.was_pressed_a():
13     vergangen = time.ticks_diff(time.ticks_ms(), start)
14     zehntel = vergangen // 100
15     anzeige = f"{zehntel // 10},{zehntel % 10} s"
16     pbc.text24(anzeige, 80, 130, pbc.YELLOW, bg=pbc.BLACK)
17     time.sleep_ms(50)
18
19 ende = time.ticks_ms()
20 dauer = time.ticks_diff(ende, start)
21 print("Start:", start, " Ende:", ende)
22 print(f"Gestoppt: {dauer} ms")
```



Terminal

```
Start: 1534  Ende: 4109
Gestoppt: 2575 ms
```

- Zeile 7 wartet auf A, Zeile 10 merkt sich den Startzeitpunkt.

- Die zweite Schleife (Zeile 12) läuft, bis A noch einmal gedrückt wird. Bei jedem Durchlauf rechnet sie die vergangene Zeit aus und zeigt sie an.
- Zeile 14 macht aus den Millisekunden Zehntelsekunden (zum Beispiel wird aus 2543 ms die Zahl 25). Zeile 15 baut daraus den Text „2,5 s“: `25 // 10` ergibt die ganzen Sekunden, `25 % 10` die Zehntel. So steht auch ein deutsches Komma in der Anzeige.
- `bg=pbc.BLACK` in Zeile 16 malt den Hintergrund hinter dem Text schwarz. Dadurch wird die alte Zahl überschrieben, ohne dass man sie vorher löschen muss.

> **Warum `ticks_diff` und nicht einfach `ende - start`?** Der Zähler kann nicht unendlich groß werden. Nach gut 12 Tagen fängt er auf dem Gerät wieder von vorn an. `ticks_diff` rechnet auch über diesen Sprung hinweg richtig.

Ausprobieren: Der Reaktionstest

Wie schnell reagierst du? Der Bildschirm ist rot. Nach einer zufälligen Wartezeit wird er grün, und du drückst so schnell wie möglich A. Wer zu früh drückt, bekommt keine Zeit. Nach drei Runden erscheint die Bestzeit.

k08_reaktionstest.py

```
 1 import pbc
 2 import time
 3 import random
 4
 5 bestzeit = 0                    # 0 heisst: noch keine Zeit
 6 pbc.clear()
 7 pbc.text24("Reaktionstest", 29, 70, pbc.WHITE)
 8 pbc.text12("Wird es gruen,", 70, 120, pbc.WHITE)
 9 pbc.text12("druecke schnell A.", 60, 138, pbc.WHITE)
10 pbc.text12("Start mit einer Taste", 44, 180, pbc.GRAY)
11 pbc.wait_for_button()
12
13 for runde in range(1, 4):
14     pbc.clear(pbc.RED)
15     pbc.text24(f"Runde {runde}", 70, 110, pbc.WHITE)
16     pbc.was_pressed_a()          # alte Tastendruecke vergessen
17     time.sleep_ms(random.randint(2000, 5000))
18     if pbc.was_pressed_a():
19         # A wurde schon waehrend der Wartezeit gedrueckt
20         pbc.clear(pbc.ORANGE)
21         pbc.text24("Zu frueh!", 65, 120, pbc.WHITE)
22         print(f"Runde {runde}: zu früh")
23     else:
24         pbc.clear(pbc.GREEN)
25         start = time.ticks_ms()
26         while not pbc.was_pressed_a():
27             time.sleep_ms(1)
28         zeit = time.ticks_diff(time.ticks_ms(), start)
29         print(f"Runde {runde}: {zeit} ms")
```

```
30          if bestzeit == 0 or zeit < bestzeit:
31              bestzeit = zeit
32          pbc.clear()
33          pbc.text24(f"{zeit} ms", 75, 120, pbc.WHITE)
34      time.sleep(2)
35
36  pbc.clear()
37  if bestzeit == 0:
38      pbc.text18("Keine gueltige Runde", 30, 120, pbc.WHITE)
39  else:
40      pbc.text18("Deine Bestzeit:", 55, 100, pbc.WHITE)
41      pbc.text24(f"{bestzeit} ms", 70, 135, pbc.YELLOW)
```







- Zeile 11 wartet, bis du bereit bist.
- Die `for`-Schleife (Zeile 13) spielt drei Runden. Die Wartezeit in Zeile 17 ist zufällig zwischen 2 und 5 Sekunden. So kann niemand den Zeitpunkt vorhersagen.
- **Schummeln erkennen:** Während `time.sleep_ms` wartet, fragt das Programm keine Tasten ab. Aber `was_pressed_a()` merkt sich jeden Druck (8.2). Ist es in Zeile 18 `True`, wurde A während der Wartezeit gedrückt: „Zu früh!“. Damit ein Druck aus der vorigen Runde nicht mitzählt, holt Zeile 16 einen gemerkten Druck vorher ab, ohne das Ergebnis zu benutzen.
- Zeilen 24 bis 28: Bildschirm grün, Startzeit merken, auf A warten, Zeit berechnen.
- **Bestzeit:** `bestzeit` beginnt mit 0, das heißt „noch keine Zeit“. In Zeile 30 wird die neue Zeit übernommen, wenn es die erste gültige ist oder wenn sie kleiner ist als die bisherige Bestzeit.

Zum Verändern: Spiele fünf Runden statt drei. Zeige am Ende zusätzlich den Durchschnitt der gültigen Runden. Gib eine Bewertung aus: unter 250 ms „Blitzschnell“, unter 400 ms „Gut“, sonst „Aufwachen!“.

Übungsaufgaben

Aufgabe 1 ⭐ Was erscheint im Terminal? Du startest das Programm und drückst nacheinander: hoch, hoch, runter, hoch, A. Schreibe auf, was im Terminal erscheint. Prüfe danach im Emulator.

k08_aufgabe1.py

```python
import pbc
import time

zahl = 10
pbc.clear()

while True:
    if pbc.was_pressed_up():
        zahl += 1
        print(zahl)
    if pbc.was_pressed_down():
        zahl -= 5
        print(zahl)
    if pbc.was_pressed_a():
        print("Ende bei", zahl)
        break
    time.sleep_ms(20)
```

Aufgabe 2 ⭐ Finde den Fehler Das Programm soll bei jedem Druck auf B einen Treffer zählen. Es zählt aber sofort los, ohne dass jemand drückt. Finde und behebe den Fehler.

k08_aufgabe2.py

```python
import pbc
import time

treffer = 0
pbc.clear()
pbc.text18("Druecke B!", 75, 120, pbc.WHITE)

while True:
    if pbc.was_pressed_b:
        treffer += 1
        print("Treffer:", treffer)
    time.sleep_ms(100)
```

Aufgabe 3 ⭐⭐ Bessere Zaubertafel Erweitere die Zaubertafel aus 8.1: (a) Mit jedem Druck auf A wechselt die Stiftfarbe der Reihe nach von Cyan über Gelb zu Magenta und wieder zu Cyan. (b) Der Stift bleibt am Rand stehen, statt aus dem Bildschirm zu laufen. Unter dem Text oben (bis y = 30) soll er auch nicht malen.

Aufgabe 4 ⭐⭐ Lichtschalter Auf dem Bildschirm ist eine graue Lampe (ein Kreis). Jeder Druck auf A schaltet sie an (gelb) oder aus (grau). *Tipp:* Speichere in einer Variablen `an` den Wert `True` oder `False`. `an = not an` schaltet um.

Aufgabe 5 ⭐⭐⭐ Taktgefühl Drücke A und zähle im Kopf bis 5. Nach genau 5 Sekunden drückst du A noch einmal. Das Programm misst die Zeit und zeigt sie an, dazu eine Bewertung: weniger als 200 ms daneben „Wie eine Uhr!“, weniger als 1000 ms daneben „Nicht schlecht“, sonst „Uebe noch!“. Achtung: Man kann zu früh oder zu spät drücken.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eingaben: `up`, `down`, `left`, `right`, `center` (Joystick drücken), `a`, `b`. Im Emulator: Pfeiltasten, Leertaste, Taste A, Taste B.
- `pbc.pressed_a()` ist `True`, **solange** A unten ist: zum Bewegen.
- `pbc.was_pressed_a()` ist `True` **einmal pro Druck** und merkt sich den Druck, bis er abgefragt wird: zum Zählen, Umschalten, Auswählen.
- `pbc.wait_for_button()` wartet auf irgendeine Taste.
  `while not pbc.was_pressed_a():` wartet auf eine bestimmte.
- Spielschleife: lesen – rechnen – zeichnen – warten.
- `time.ticks_ms()` ist ein Millisekundenzähler. `time.ticks_diff(spaeter, frueher)` liefert die vergangene Zeit.

KAPITEL 9

LEDs und Töne

Worum geht's? Der PicoBoy kann mehr als nur Bilder zeigen: Er hat drei kleine Lämpchen, eine Lampe, die in jeder Farbe leuchtet, und einen Lautsprecher. Du schaltest Lichter, mischst Farben, spielst Töne und programmierst eine kleine Melodie. Am Ende baust du ein Klavier, das du mit den Tasten spielst.

9.1 Die drei Status-LEDs

Links neben dem Bildschirm sitzen drei **LEDs** (Leuchtdioden): rot, gelb und grün. Jede kannst du einzeln ein- und ausschalten:

Befehl	Wirkung
`pbc.led_red(True)`	rote LED an
`pbc.led_red(False)`	rote LED aus
`pbc.led_yellow(…)`, `pbc.led_green(…)`	dasselbe für Gelb und Grün
`pbc.leds_off()`	alle drei aus

Im Emulator siehst du die LEDs am linken Rand des Geräts. Das folgende Programm schaltet sie nacheinander ein und wieder aus:

k09_leds.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text18("Status-LEDs", 70, 125, pbc.WHITE)
 6
 7 pbc.led_red(True)
 8 time.sleep(1)
 9 pbc.led_yellow(True)
10 time.sleep(1)
11 pbc.led_green(True)
12 time.sleep(2)
13
14 pbc.led_yellow(False)
15 time.sleep(1)
16 pbc.leds_off()
```



- Zeile 7 schaltet die rote LED ein. Danach wartet das Programm eine Sekunde.
- In Zeile 9 und 11 kommen Gelb und Grün dazu. Eine LED bleibt an, bis du sie ausschaltest. Nach Zeile 11 leuchten also alle drei (Bild).
- Zeile 14 schaltet nur die gelbe LED aus, Zeile 16 alle, die noch an sind.

In den Klammern steht ein **Wahrheitswert**: `True` heißt an, `False` heißt aus. Lässt du die Klammer leer, gilt `True`: `pbc.led_red()` schaltet die LED also ein.

> **Licht bleibt an.** Was ein Programm einschaltet, leuchtet weiter, auch wenn das Programm zu Ende ist. Im Emulator gehen die LEDs erst beim nächsten ▶ **Testen** oder bei ■ **Stop** aus, auf dem Gerät nicht einmal dann. Schalte deshalb am Ende alles mit `pbc.leds_off()` aus.

Tasten schalten Licht. Weil `True` und `False` auch das Ergebnis einer Bedingung sein können, kannst du eine Taste direkt mit einer LED verbinden:

k09_tasten_leds.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text12("A = rot, B = gruen", 20, 100, pbc.WHITE)
 6 pbc.text12("Joystick = gelb", 20, 120, pbc.WHITE)
 7
 8 while True:
 9     pbc.led_red(pbc.pressed_a())
10     pbc.led_green(pbc.pressed_b())
11     pbc.led_yellow(pbc.pressed_up() or pbc.pressed_down())
12     time.sleep_ms(20)
```

`pbc.pressed_a()` liefert `True`, solange A gedrückt ist, sonst `False`. Zeile 9 gibt diesen Wert direkt an `pbc.led_red` weiter: Die rote LED leuchtet genau so lange, wie du A drückst. In Zeile 11 entscheidet ein `or`: Gelb leuchtet, wenn der Joystick nach oben **oder** nach unten gedrückt ist.

9.2 Die RGB-LED

Im Pluszeichen des Schriftzugs „PBC+“ sitzt eine besondere LED: die **RGB-LED**. Sie enthält drei winzige Lampen in Rot, Grün und Blau. Für jede gibst du an, wie hell sie leuchten soll, von 0 (aus) bis 255 (ganz hell):

```
pbc.rgb_led(rot, gruen, blau)
```

Die drei Farben mischen sich zu einer neuen Farbe. Das funktioniert wie bei den Farben auf einem Bildschirm, nicht wie beim Malen mit dem Pinsel: Rot und Grün ergeben **Gelb**, alle drei zusammen ergeben **Weiß**.

`rot, gruen, blau`	Farbe	`rot, gruen, blau`	Farbe
`255, 0, 0`	Rot	`255, 255, 0`	Gelb
`0, 255, 0`	Grün	`255, 0, 255`	Magenta
`0, 0, 255`	Blau	`0, 255, 255`	Cyan
`255, 255, 255`	Weiß	`255, 100, 0`	Orange

k09_rgb.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text18("RGB-LED", 80, 125, pbc.WHITE)
 6
 7 pbc.rgb_led(255, 0, 0)      # rot
 8 time.sleep(1)
 9 pbc.rgb_led(255, 255, 0)    # gelb = rot + gruen
10 time.sleep(1)
11 pbc.rgb_led(30, 30, 30)     # schwaches Weiss
12 time.sleep(1)
13 pbc.rgb_led_off()
```

Jede Zeile mit `pbc.rgb_led` ersetzt die vorige Farbe. Zeile 11 mischt alle drei Farben, aber nur schwach (30 von 255): Das ergibt ein gedämpftes Weiß. `pbc.rgb_led_off()` schaltet die RGB-LED aus; das ist dasselbe wie `pbc.rgb_led(0, 0, 0)`.

> Die Zahlen für `pbc.rgb_led` sind dieselben wie für `pbc.rgb(r, g, b)` aus Kapitel 5. Damit kannst du die LED und den Bildschirm in derselben Farbe leuchten lassen. Nimm nur ganze Zahlen von 0 bis 255.

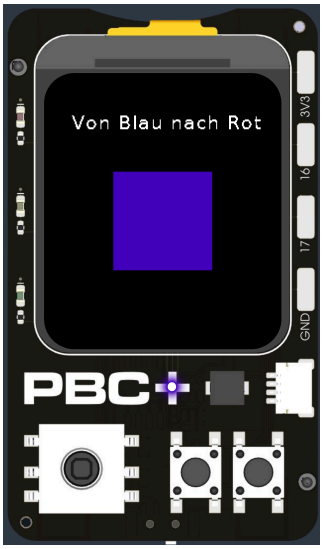
Ein Farbverlauf. Die Schleife zählt `rot` in Fünferschritten von 0 bis 255. Gleichzeitig wird `blau` immer kleiner. So wandert die Farbe langsam von Blau über Lila nach Rot, auf der LED und im Quadrat auf dem Bildschirm:

k09_farbverlauf.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text18("Von Blau nach Rot", 28, 40, pbc.WHITE)
 6
 7 for rot in range(0, 256, 5):
 8     blau = 255 - rot
 9     pbc.rgb_led(rot, 0, blau)
10     pbc.fill_rect(70, 100, 100, 100, pbc.rgb(rot, 0, blau))
11     time.sleep_ms(40)
12
13 time.sleep(1)
14 pbc.rgb_led_off()
```

- Zeile 7 läuft mit `range(0, 256, 5)` über 0, 5, 10, … bis 255.
- Zeile 8 rechnet das passende Blau aus: Ist `rot` 0, ist `blau` 255 und umgekehrt.
- Zeile 9 setzt die LED, Zeile 10 malt ein Quadrat in genau dieser Farbe.

Im Bild ist der Verlauf gerade bei einem bläulichen Lila angekommen.



9.3 Töne: `pbc.tone` und `pbc.tone_off`

Ein Ton entsteht, wenn der Lautsprecher sehr schnell hin und her schwingt. Wie oft er das pro Sekunde tut, heißt **Frequenz** und wird in **Hertz** (Hz) gemessen. Je höher die Frequenz, desto höher der Ton. Der Ton a, nach dem Musiker ihre Instrumente stimmen, hat 440 Hz. Menschen hören ungefähr von 20 Hz bis 16 000 Hz; angenehm klingen auf dem kleinen Lautsprecher Töne zwischen etwa 200 und 2000 Hz.

Es gibt zwei Arten, einen Ton zu spielen:

Befehl	Wirkung
`pbc.tone(440)`	**Dauerton** mit 440 Hz, klingt, bis `pbc.tone_off()` kommt
`pbc.tone(440, 500)`	Ton mit 440 Hz, hört nach 500 ms von selbst auf
`pbc.tone_off()`	Ton sofort ausschalten

k09_ton.py

```
 1 import pbc
 2 import time
 3
 4 # Dauerton: klingt, bis tone_off kommt
 5 pbc.tone(440)
 6 time.sleep(1)
 7 pbc.tone_off()
 8
 9 time.sleep_ms(500)
10
11 # Ton mit Dauer: hoert nach 500 ms von selbst auf
12 pbc.tone(880, 500)
13 time.sleep_ms(500)
```

- Zeile 5 startet einen Dauerton. Zeile 6 wartet eine Sekunde, solange klingt der Ton. Zeile 7 schaltet ihn aus.

- Zeile 9 macht eine halbe Sekunde Pause, in der nichts zu hören ist.
- Zeile 12 spielt einen Ton, der doppelt so hoch ist (880 Hz) und nach 500 ms von selbst aufhört.
- Zeile 13 wartet, bis der Ton fertig ist. Am Programmende wäre das nicht unbedingt nötig, sobald danach noch ein Ton kommt, aber schon.

`pbc.tone` wartet nicht. Der Befehl schaltet den Ton nur ein und das Programm läuft sofort weiter. Das ist praktisch: In einem Spiel kann ein Ton klingen, während sich die Figur weiterbewegt. Für Töne nacheinander musst du aber selbst warten:

k09_fehler_ohne_pause.py

```
1 import pbc
2
3 pbc.tone(262, 400)
4 pbc.tone(330, 400)
5 pbc.tone(392, 400)
```

Du erwartest drei Töne, hörst aber nur den letzten: Die drei Zeilen laufen in einem winzigen Augenblick ab, und jeder neue Ton ersetzt den vorigen sofort. Der Lautsprecher kann immer nur **einen** Ton spielen. Hinter jedes `pbc.tone(f, ms)` gehört deshalb ein `time.sleep_ms(ms)`.

Ein Dauerton hört nicht von allein auf, auch nicht am Ende des Programms. Er klingt weiter, bis `pbc.tone_off()` kommt oder du auf ■ **Stop** klickst. Schreibe zu jedem `pbc.tone(f)` ohne Dauer auch ein `pbc.tone_off()`.

9.4 Eine Melodie

Eine Melodie ist eine Folge von Tönen mit bestimmter Höhe und Länge. Diese Frequenzen gehören zu den Tönen der C-Dur-Tonleiter:

Ton	c	d	e	f	g	a	h	c’
Hz	262	294	330	349	392	440	494	523

Damit spielst du den Anfang von „Alle meine Entchen“:

k09_entchen.py

```
1  import pbc
2  import time
3
4  # Frequenzen der Noten in Hz
5  C = 262
6  D = 294
7  E = 330
8  F = 349
9  G = 392
10
11 VIERTEL = 300    # Dauer einer Viertelnote in ms
12 HALBE = 600      # eine halbe Note ist doppelt so lang
13
14 pbc.tone(C, VIERTEL - 50)
```

```
15 time.sleep_ms(VIERTEL)
16 pbc.tone(D, VIERTEL - 50)
17 time.sleep_ms(VIERTEL)
18 pbc.tone(E, VIERTEL - 50)
19 time.sleep_ms(VIERTEL)
20 pbc.tone(F, VIERTEL - 50)
21 time.sleep_ms(VIERTEL)
22 pbc.tone(G, HALBE - 50)
23 time.sleep_ms(HALBE)
24 pbc.tone(G, HALBE - 50)
25 time.sleep_ms(HALBE)
```

- Zeile 5 bis 9 geben den Tönen Namen. So liest sich `pbc.tone(E, …)` leichter als `pbc.tone(330, …)`. Weil sich diese Werte nie ändern, sind die Namen großgeschrieben (Konstanten).
- `VIERTEL` und `HALBE` legen fest, wie lang eine Note dauert. Änderst du nur Zeile 11 und 12, spielt das ganze Lied schneller oder langsamer.
- Jede Note besteht aus zwei Zeilen: Ton starten, dann warten. Der Ton ist 50 ms **kürzer** als die Wartezeit. Diese kleine Lücke trennt die Noten. Ohne sie würden die beiden G am Ende zu einem langen Ton verschmelzen.

> Die Melodie schreibt man hier noch Note für Note hin. In Kapitel 10 lernst du, eine Melodie als Text wie `"cdefgg"` zu speichern, und in Kapitel 13 mit Listen. Dann wird das Programm viel kürzer.

Töne ausrechnen. Von einem Ton zum nächsten Halbton steigt die Frequenz immer um denselben Faktor: Sie wird mit $2^{1/12} \approx 1{,}06$ malgenommen. Nach 12 Halbtönen hat sie sich genau verdoppelt. Das kann Python ausrechnen:

k09_tonleiter.py

```
1 import pbc
2 import time
3
4 # 13 Halbtoene von a (440 Hz) bis zum naechsten a
5 for schritt in range(13):
6     frequenz = int(440 * 2 ** (schritt / 12))
7     print(schritt, frequenz)
8     pbc.tone(frequenz, 200)
9     time.sleep_ms(250)
```

Terminal

```
0 440
1 466
2 493
3 523
4 554
5 587
6 622
7 659
8 698
9 739
10 783
11 830
12 880
```

`2 ** (schritt / 12)` ist der Faktor für `schritt` Halbtöne. Das Ergebnis ist eine Kommazahl. `int()` macht daraus eine ganze Zahl, indem es die Nachkommastellen abschneidet.

Gib `pbc.tone` immer eine **ganze Zahl**. Mit einer Kommazahl bricht das Programm ab:

k09_fehler_komma.py

```
1 import pbc
2
3 frequenz = 440 * 2 ** (1 / 12)
4 pbc.tone(frequenz, 200)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: can't convert float to int
```

Berechnete Frequenzen wandelst du deshalb mit `int()` um. Dasselbe gilt für die Zahlen bei `pbc.rgb_led`.

Ausprobieren: Das Mini-Klavier

Jede Taste spielt einen Ton, solange du sie drückst. Die grüne LED zeigt an, dass gerade ein Ton klingt.

k09_klavier.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text18("Mini-Klavier", 70, 30, pbc.WHITE)
 6 pbc.text12("links: c", 40, 90, pbc.YELLOW)
 7 pbc.text12("hoch: d", 140, 90, pbc.YELLOW)
 8 pbc.text12("rechts: e", 40, 115, pbc.YELLOW)
 9 pbc.text12("runter: f", 140, 115, pbc.YELLOW)
10 pbc.text12("Mitte: g", 40, 140, pbc.YELLOW)
11 pbc.text12("A: a    B: h", 140, 140, pbc.YELLOW)
12
13 alter_ton = 0
14 while True:
15     # 1. Lesen: Welche Taste ist unten?
16     ton = 0
17     if pbc.pressed_left():
18         ton = 262
19     elif pbc.pressed_up():
20         ton = 294
21     elif pbc.pressed_right():
22         ton = 330
23     elif pbc.pressed_down():
24         ton = 349
25     elif pbc.pressed_center():
26         ton = 392
```



```
27      elif pbc.pressed_a():
28          ton = 440
29      elif pbc.pressed_b():
30          ton = 494
31
32      # 2. Nur etwas tun, wenn sich der Ton geaendert hat
33      if ton != alter_ton:
34          if ton == 0:
35              pbc.tone_off()
36          else:
37              pbc.tone(ton)
38          pbc.led_green(ton > 0)
39          alter_ton = ton
40      time.sleep_ms(20)
```

- Zeile 4 bis 11 zeichnen die Anleitung. Sie steht in zwei Spalten, die bei x = 40 und x = 140 beginnen.
- Die Spielschleife folgt dem Muster aus Kapitel 8: lesen, rechnen, „zeichnen“ (hier: Ton und LED schalten), warten.
- Zeile 16 bis 30 lesen die Tasten. `ton` ist 0, wenn keine Taste gedrückt ist. Durch das `elif` zählt immer nur die erste gedrückte Taste.
- Zeile 33 vergleicht mit dem Ton aus dem letzten Durchlauf. Nur wenn sich etwas geändert hat, wird der Lautsprecher umgeschaltet. So bekommt der Lautsprecher nicht 50-mal pro Sekunde denselben Befehl.
- Zeile 38 schaltet die grüne LED mit einem Vergleich: `ton > 0` ist `True`, solange ein Ton klingt.

Zum Verändern: Lege andere Töne auf die Tasten, zum Beispiel eine Tonleiter eine Oktave höher (doppelte Frequenzen). Lass die RGB-LED bei jedem Ton in einer anderen Farbe leuchten. Schaffst du es, „Alle meine Entchen“ auf deinem Klavier zu spielen?

Übungsaufgaben

Aufgabe 1 ⭐ Welche Lichter brennen? Lies das Programm, ohne es zu testen. Welche LEDs leuchten nach einer halben Sekunde? Welche leuchten noch, wenn das Programm zu Ende ist, und in welcher Farbe leuchtet dann die RGB-LED? Prüfe danach im Emulator.

k09_aufgabe1.py

```
1 import pbc
2 import time
3
4 pbc.led_red()
5 pbc.led_green(True)
6 pbc.rgb_led(0, 0, 255)
7 time.sleep(1)
8 pbc.led_red(False)
```

```
 9 pbc.led_yellow(True)
10 pbc.rgb_led(255, 255, 0)
11 time.sleep(1)
12 pbc.led_green(False)
```

Aufgabe 2 ⭐ Finde die Fehler Das Programm soll ein Signal spielen: zweimal kurz, einmal lang. Es gibt keine Fehlermeldung, trotzdem stimmt etwas nicht. Was hörst du? Korrigiere das Programm.

k09_aufgabe2.py

```
1 import pbc
2 import time
3
4 # Signal: kurz - kurz - lang
5 pbc.tone(880, 150)
6 pbc.tone(880, 150)
7 time.sleep_ms(250)
8 pbc.tone(660, 1000)
9 time.sleep_ms(1000)
```

Aufgabe 3 ⭐⭐ Die Entchen schwimmen weiter Ergänze `k09_entchen.py` um die zweite Zeile des Lieds: „schwimmen auf dem See“ ist viermal der Ton a (Viertel), dann g, so lang wie zwei halbe Noten. Tipp: Für die vier gleichen Töne passt eine `for`-Schleife.

Aufgabe 4 ⭐⭐ Blaulicht Schreibe ein Programm für ein Einsatzfahrzeug: Die RGB-LED wechselt fünfmal zwischen Blau und Rot, je eine halbe Sekunde. Bei Blau klingt ein Ton mit 440 Hz, bei Rot einer mit 587 Hz. Auf dem Bildschirm steht „Einsatz!“. Am Ende sind Ton und Licht aus.

Aufgabe 5 ⭐⭐⭐ Ein Metronom Ein Metronom gibt beim Musizieren den Takt vor. Schreibe ein Programm, das gleichmäßig klickt (kurzer Ton mit 1000 Hz) und bei jedem Klick kurz die rote LED aufleuchten lässt. Das Tempo steht in Schlägen pro Minute auf dem Bildschirm (Start: 100). Mit dem Joystick nach oben wird es um 10 schneller, nach unten um 10 langsamer. Wie viele Millisekunden liegen bei 100 Schlägen pro Minute zwischen zwei Klicks?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `pbc.led_red(True)` / `pbc.led_red(False)` schaltet eine Status-LED an und aus, ebenso `led_yellow` und `led_green`; `pbc.leds_off()` alle.
- `pbc.rgb_led(r, g, b)` mischt eine Farbe aus Rot, Grün und Blau (je 0–255), `pbc.rgb_led_off()` schaltet die RGB-LED aus.
- LEDs bleiben an, bis du sie ausschaltest, auch nach dem Programmende.
- `pbc.tone(f)` spielt einen Dauerton mit `f` Hertz bis `pbc.tone_off()`; `pbc.tone(f, ms)` hört nach `ms` Millisekunden von selbst auf.
- `pbc.tone` wartet nicht. Für eine Melodie nach jedem Ton `time.sleep_ms(…)`; ein neuer Ton ersetzt den alten.
- Frequenzen sind ganze Zahlen; höhere Frequenz = höherer Ton, doppelte Frequenz = eine Oktave höher.

KAPITEL 10

Texte (Strings)

Worum geht's? Bisher hast du Texte nur ausgegeben. Jetzt nimmst du sie auseinander: Du zählst Zeichen, holst einzelne Buchstaben und Teilstücke heraus, wandelst Groß- in Kleinbuchstaben um und rechnest mit Buchstaben. Damit baust du am Ende eine Geheimschrift, mit der schon Julius Caesar Nachrichten verschlüsselt hat.

10.1 Zeichen zählen und einzelne Zeichen holen: `len`, `text[i]`

Ein String ist eine Folge von **Zeichen**: Buchstaben, Ziffern, Leerzeichen, Satzzeichen. `len(text)` liefert die **Länge**, also die Anzahl der Zeichen. Mit eckigen Klammern holst du ein einzelnes Zeichen heraus. Die Zahl in den Klammern heißt **Index** (Position). Wichtig: Python zählt ab **0**.

Zeichen	`P`	`y`	`t`	`h`	`o`	`n`
Index von vorn	0	1	2	3	4	5
Index von hinten	−6	−5	−4	−3	−2	−1

k10_laenge.py

```
1 wort = "Python"
2 print(len(wort))
3 print(wort[0], wort[1], wort[5])
4 print(wort[-1], wort[-2])
5
6 name = input("Wie heißt du? ")
7 print(f"Dein Name hat {len(name)} Zeichen.")
8 print(f"Er beginnt mit {name[0]} und endet mit {name[-1]}.")
```

Terminal

```
6
P y n
n o
Wie heißt du? Malte
Dein Name hat 5 Zeichen.
Er beginnt mit M und endet mit e.
```

- `"Python"` hat 6 Zeichen. Das erste hat den Index 0, das letzte den Index 5, also **Länge minus 1**.
- Ein negativer Index zählt von hinten: `wort[-1]` ist immer das letzte Zeichen, egal wie lang der Text ist. Das nutzt Zeile 8 für den Namen, dessen Länge vorher niemand kennt.
- Auch Leerzeichen und Satzzeichen sind Zeichen und werden mitgezählt.

Einen Index, den es nicht gibt, kann Python nicht holen. `"Python"` hat die Indizes 0 bis 5; `wort[6]` führt zu einem Fehler:

k10_fehler_index.py

```
1 wort = "Python"
2 print(wort[6])
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: string index out of range
```

`IndexError: string index out of range` heißt: Der Index liegt außerhalb des Textes. Meist ist man um eins verrutscht, weil das Zählen bei 0 beginnt.

10.2 Teilstücke: `text[a:b]`

Mit einem Doppelpunkt holst du ein ganzes Stück aus einem Text heraus. Man nennt das **Slicing** (Abschneiden). `text[a:b]` beginnt beim Index `a` und hört **vor** dem Index `b` auf:

k10_teile.py

```
 1 datum = "05.10.2026"
 2 tag = datum[0:2]
 3 monat = datum[3:5]
 4 jahr = datum[6:10]
 5 print("Tag:", tag, "Monat:", monat, "Jahr:", jahr)
 6
 7 wort = "Fussball"
 8 print(wort[:4])
 9 print(wort[4:])
10 print(wort[-4:])
```

Terminal

```
Tag: 05 Monat: 10 Jahr: 2026
Fuss
ball
ball
```

- `datum[0:2]` sind die Zeichen mit Index 0 und 1, also `"05"`. Das Zeichen mit Index 2 (der Punkt) ist nicht mehr dabei.
- Die Länge des Stücks ist `b - a`: `datum[3:5]` hat 2 Zeichen.
- Lässt du `a` weg, geht es ab dem Anfang: `wort[:4]` sind die ersten vier Zeichen. Lässt du `b` weg, geht es bis zum Ende: `wort[4:]`.
- Auch hier zählt ein negativer Index von hinten: `wort[-4:]` sind die letzten vier Zeichen.

Beim Slicing gibt es keinen `IndexError`. Reicht das Stück über das Ende hinaus, bekommst du einfach weniger Zeichen: `"abc"[1:10]` ist `"bc"`.

10.3 Nützliche Methoden: `upper`, `lower`, `replace`, `in`

Strings bringen eigene Funktionen mit. Man schreibt sie mit einem Punkt hinter den Text: `satz.upper()`. Solche Funktionen, die zu einem Wert gehören, heißen **Methoden**.

Methode	Ergebnis
`text.upper()`	Text in Großbuchstaben
`text.lower()`	Text in Kleinbuchstaben
`text.replace(alt, neu)`	Text, in dem jedes `alt` durch `neu` ersetzt ist
`teil in text`	`True`, wenn `teil` irgendwo in `text` vorkommt

k10_methoden.py

```
1 satz = "Der PicoBoy spielt Musik"
2 print(satz.upper())
3 print(satz.lower())
4 print(satz.replace("Musik", "Tetris"))
5 print(satz)
6 print("Boy" in satz, "boy" in satz)
7
8 antwort = input("Noch eine Runde? ")
9 if antwort.lower() == "ja":
10     print("Los geht's!")
11 else:
12     print("Tschüss!")
```

Terminal

```
DER PICOBOY SPIELT MUSIK
der picoboy spielt musik
Der PicoBoy spielt Tetris
Der PicoBoy spielt Musik
True False
Noch eine Runde? JA
Los geht's!
```

- Zeile 5 zeigt: `satz` selbst ist unverändert. Die Methoden liefern einen **neuen** Text. Willst du das Ergebnis behalten, speichere es: `satz = satz.upper()`.
- `in` unterscheidet Groß- und Kleinschreibung: `"boy"` kommt nicht vor.
- Zeile 9 ist ein häufiger Trick: Durch `lower()` ist es egal, ob jemand `ja`, `Ja` oder `JA` eintippt.

Strings kann man nicht verändern. Ein einzelnes Zeichen kannst du lesen, aber nicht ersetzen:

k10_fehler_aendern.py

```
1 wort = "Hund"
2 wort[0] = "M"
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'str' object doesn't support item assignment
```

Baue stattdessen einen neuen Text zusammen: `wort = "M" + wort[1:]` ergibt `"Mund"`.

In MicroPython ändern `upper()` und `lower()` nur die Buchstaben A bis Z. Umlaute und ß bleiben, wie sie sind (das „große“ Python auf dem Computer macht hier aus ü ein Ü):

k10_umlaut_upper.py

```
1 print("Grüße aus Köln".upper())
2 print("ÄRGER".lower())
```

Terminal

```
GRüßE AUS KöLN
Ärger
```

10.4 Texte Zeichen für Zeichen durchlaufen

Die `for`-Schleife kann nicht nur Zahlen durchlaufen, sondern auch die Zeichen eines Textes, eins nach dem anderen:

k10_vokale.py

```
 1 satz = input("Gib einen Satz ein: ")
 2
 3 anzahl = 0
 4 for zeichen in satz:
 5     if zeichen in "aeiouAEIOU":
 6         anzahl += 1
 7 print(f"Der Satz hat {anzahl} Vokale.")
 8
 9 for i in range(len(satz)):
10     print(i, satz[i])
```

Terminal

```
Gib einen Satz ein: Ein Uhu
Der Satz hat 4 Vokale.
0 E
1 i
2 n
3
4 U
5 h
6 u
```

- In Zeile 4 bekommt `zeichen` bei jedem Durchlauf das nächste Zeichen des Satzes: erst `"E"`, dann `"i"`, dann `"n"` …

- Zeile 5 prüft mit `in`, ob das Zeichen einer der Vokale in `"aeiouAEIOU"` ist. Wenn ja, wird gezählt.
- Brauchst du auch die Position, läuft die Schleife über die Indizes: `range(len(satz))` liefert 0, 1, 2, … bis Länge minus 1 (Zeile 9 und 10).

Schreibmaschine. Zeichen für Zeichen geht auch auf dem Bildschirm. Jeder Buchstabe erscheint mit einem kurzen Klick:

k10_schreibmaschine.py

```
 1 import pbc
 2 import time
 3
 4 text = "Hallo, ich bin dein PBC+!"
 5
 6 pbc.clear()
 7 x = 20
 8 for zeichen in text:
 9     pbc.text(zeichen, x, 130, pbc.GREEN)
10     if zeichen != " ":
11         pbc.tone(2000, 10)
12     x += 8
13     time.sleep_ms(120)
```



Die Schrift `pbc.text` ist für jedes Zeichen genau 8 Pixel breit. Deshalb rückt Zeile 12 die Position `x` nach jedem Zeichen um 8 weiter. Bei Leerzeichen bleibt es still (Zeile 10).

10.5 Zeichen und Zahlen: `ord` und `chr`

Im Computer ist jedes Zeichen eine Zahl, sein **Zeichencode**. `ord(zeichen)` liefert den Code eines Zeichens, `chr(zahl)` macht aus einem Code wieder ein Zeichen. Die Großbuchstaben A bis Z haben die Codes 65 bis 90, die kleinen a bis z die Codes 97 bis 122, das Leerzeichen 32.

k10_ord_chr.py

```
1 print(ord("A"), ord("B"), ord("Z"))
2 print(ord("a"), ord("z"), ord(" "))
3 print(chr(72) + chr(105))
4
5 alphabet = ""
6 for code in range(65, 91):
7     alphabet += chr(code)
8 print(alphabet)
9 print(chr(ord("C") + 1))
```

Terminal

```
65 66 90
97 122 32
Hi
ABCDEFGHIJKLMNOPQRSTUVWXYZ
D
```

- Zeile 6 und 7 bauen das Alphabet zusammen: `alphabet += chr(code)` hängt bei jedem Durchlauf einen Buchstaben hinten an. Der Text beginnt leer (`""`, Zeile 5).
- Zeile 9 rechnet mit einem Buchstaben: Code von C plus 1 ergibt den Code von D.

Weil Zeichen Codes haben, kann Python sie auch vergleichen: `"A" < "B"` ist `True`, und `"A" <= zeichen <= "Z"` prüft, ob `zeichen` ein Großbuchstabe ist. Das ist eine Kurzform für `"A" <= zeichen and zeichen <= "Z"`.

10.6 Umlaute auf dem Display

In der Konsole funktionieren Umlaute, auf dem Display nicht:

k10_umlaute.py

```
 1 import pbc
 2
 3 text = "Schöne Grüße"
 4 print(text, len(text))
 5
 6 pbc.clear()
 7 pbc.text18(text, 20, 60, pbc.WHITE)
 8
 9 ohne = text.replace("ö", "oe").replace("ü", "ue").replace("ß", "ss")
10 print(ohne, len(ohne))
11 pbc.text18(ohne, 20, 120, pbc.GREEN)
```

Terminal

```
Schöne Grüße 12
Schoene Gruesse 15
```



Warum erscheinen für ö zwei Fragezeichen und für ü auch? Die Schriften des Displays kennen nur die Zeichen mit den Codes 32 bis 127: Buchstaben ohne Umlaute, Ziffern und Satzzeichen. Wird ein Text an das Display geschickt, zerlegt Python ihn in **Bytes** (kleine Zahlen von 0 bis 255). Ein normaler Buchstabe ist ein Byte, ein Umlaut oder ß aber **zwei** Bytes, das Eurozeichen € sogar drei. Jedes Byte, das die Schrift nicht kennt, wird als „?“ gezeichnet. Aus „Grüße“ wird deshalb „Gr????e“.

Für Python selbst ist ü trotzdem **ein** Zeichen: `len("Schöne Grüße")` ist 12. Zeile 9 ersetzt die Umlaute, bevor der Text auf das Display kommt. Die drei `replace` hängen aneinander: Jedes arbeitet mit dem Ergebnis des vorigen weiter.

Ausprobieren: Die Geheimschrift

Julius Caesar soll geheime Nachrichten so verschlüsselt haben: Jeder Buchstabe wird im Alphabet um eine feste Zahl von Stellen verschoben, den **Schlüssel**. Mit dem Schlüssel 3 wird aus A ein D, aus B ein E, … und am Ende geht es vorn weiter: Aus X wird A, aus Y wird B, aus Z wird C.

k10_caesar.py

```
 1 import pbc
 2
 3 text = input("Text: ")
 4 text = text.upper()
 5 schluessel = int(input("Verschiebung (1-25): "))
 6
 7 geheim = ""
 8 for zeichen in text:
 9     if "A" <= zeichen <= "Z":
10         nummer = ord(zeichen) - ord("A")         # A=0, B=1 ... Z=25
11         nummer = (nummer + schluessel) % 26      # verschieben, Z -> A
12         geheim += chr(nummer + ord("A"))
13     else:
14         geheim += zeichen                        # Leerzeichen usw.
15
16 print("Geheim:", geheim)
17
18 pbc.clear()
19 pbc.text18("Geheimbotschaft", 30, 40, pbc.YELLOW)
20 pbc.text(text, 10, 110, pbc.WHITE)
21 pbc.text(geheim, 10, 150, pbc.GREEN)
22 pbc.text(f"Schluessel: {schluessel}", 10, 200, pbc.GRAY)
```

Terminal

```
Text: Treffpunkt am Brunnen
Verschiebung (1-25): 3
Geheim: WUHIISXQNW DP EUXQQHQ
```



- Zeile 4 macht alles groß. So muss das Programm nur A bis Z verschlüsseln.
- Zeile 9 prüft, ob das Zeichen ein Großbuchstabe ist. Leerzeichen, Ziffern und Satzzeichen bleiben unverändert (Zeile 14).
- Zeile 10 macht aus dem Buchstaben eine Nummer von 0 bis 25: `ord("D") - ord("A")` ist 3.
- Zeile 11 verschiebt. Der Rest bei der Division durch 26 (`% 26`) sorgt dafür, dass es nach Z wieder bei A weitergeht: Z hat die Nummer 25, 25 + 3 = 28, und 28 % 26 = 2, das ist C.
- Zeile 12 rechnet zurück in einen Buchstaben und hängt ihn an `geheim` an.

Zum Verändern: Zum Entschlüsseln verschiebst du in die andere Richtung. Probiere das Programm mit dem Schlüssel `-3` und dem Text `WUHIISXQNW DP EUXQQHQ`. Was kommt heraus? Erweitere das Programm so, dass es auch Kleinbuchstaben verschlüsselt, ohne sie vorher groß zu machen.

Übungsaufgaben

Aufgabe 1 ⭐ **Was gibt das Programm aus?** Schreibe die fünf Zeilen der Ausgabe auf, ohne das Programm zu testen. Prüfe danach.

k10_aufgabe1.py

```
1 stadt = "Hamburg"
2 print(len(stadt))
3 print(stadt[1] + stadt[-1])
4 print(stadt[0:3].upper())
5 print(stadt.replace("burg", "mer"))
6 print("b" in stadt, "B" in stadt)
```

Aufgabe 2 ⭐ **Finde die Fehler** Das Programm soll den letzten Buchstaben von „banane“ ausgeben und das Wort danach mit großem B schreiben. Es hat zwei Fehler. Lies die Fehlermeldungen und korrigiere sie.

k10_aufgabe2.py

```
1 frucht = "banane"
2 letztes = frucht[6]
3 print("Letzter Buchstabe:", letztes)
4 frucht[0] = "B"
5 print(frucht)
```

Aufgabe 3 ⭐⭐ **Rückwärts und Palindrome** Schreibe ein Programm, das ein Wort einliest und es rückwärts ausgibt. Tipp: Durchlaufe das Wort und setze jedes Zeichen **vor** das bisherige Ergebnis. Prüfe dann, ob das Wort ein Palindrom ist, also rückwärts gleich lautet („Rentner“, „Anna“, „Lagerregal“). Groß- und Kleinschreibung soll keine Rolle spielen.

Aufgabe 4 ⭐⭐ **Melodie als Text** In Kapitel 9 hast du jede Note einzeln programmiert. Speichere eine Melodie jetzt als Text, zum Beispiel `"cdefgg-aaaag---"`, wobei `-` eine Pause ist. Durchlaufe den Text und spiele für jeden Buchstaben den passenden Ton (Frequenzen in Abschnitt 9.4).

Aufgabe 5 ⭐⭐ **Benutzername** Lies Vor- und Nachnamen ein. Der Benutzername besteht aus den ersten drei Buchstaben des Vornamens und den ersten drei des Nachnamens, alles klein. Aus Lina Schröder wird `linsch`.

Aufgabe 6 ⭐⭐⭐ Den Code knacken Du hast die Nachricht `WUHIISXQNW DP EUXQQHQ` abgefangen, kennst aber den Schlüssel nicht. Schreibe ein Programm, das alle Schlüssel von 1 bis 25 ausprobiert und jedes Ergebnis mit dem Schlüssel ausgibt. Welche Zeile ergibt einen Sinn? Was sagt das über die Sicherheit der Caesar-Verschlüsselung?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `len(text)` zählt die Zeichen. `text[i]` holt das Zeichen mit dem Index `i`; gezählt wird ab 0, `text[-1]` ist das letzte Zeichen.
- `text[a:b]` ist das Stück von Index `a` bis **vor** `b`; `text[:b]` vom Anfang, `text[a:]` bis zum Ende.
- `upper()`, `lower()` und `replace(alt, neu)` liefern einen **neuen** Text; Strings selbst kann man nicht ändern. `teil in text` prüft, ob etwas vorkommt.
- `for zeichen in text:` durchläuft einen Text Zeichen für Zeichen; `text += zeichen` hängt etwas an.
- `ord("A")` ist 65, `chr(65)` ist `"A"`. Mit Codes kann man rechnen.
- Umlaute erscheinen auf dem Display als „??“ (ein „?“ pro Byte); in der Konsole gehen sie. Vor dem Anzeigen mit `replace` ersetzen.

KAPITEL 11

Funktionen

Worum geht's? Deine Programme werden länger, und manche Teile schreibst du immer wieder fast gleich. Mit einer eigenen **Funktion** gibst du so einem Teil einen Namen und rufst ihn danach so oft auf, wie du willst, jedes Mal mit anderen Werten. So werden Programme kürzer und übersichtlicher.

11.1 Eigene Funktionen mit `def`

Funktionen kennst du schon: `print`, `input`, `pbc.clear`, `pbc.text24`. Jemand hat sie geschrieben, und du rufst sie mit ihrem Namen und runden Klammern auf. Jetzt schreibst du eigene Funktionen.

k11_ansage.py

```
1 def ansage():
2     print("*** Achtung! ***")
3     print("Der Bus fährt in 5 Minuten.")
4     print()
5
6
7 print("Programmstart")
8 ansage()
9 ansage()
10 print("Programmende")
```

Terminal

```
Programmstart
*** Achtung! ***
Der Bus fährt in 5 Minuten.

*** Achtung! ***
Der Bus fährt in 5 Minuten.

Programmende
```

Zeile 1 bis 4 **definieren** die Funktion `ansage`:

- `def` steht für *define*, „festlegen". Dahinter kommt der Name der Funktion, dann runde Klammern und ein **Doppelpunkt**.
- Die eingerückten Zeilen darunter (hier Zeile 2 bis 4) sind der **Funktionskörper**. Sie gehören zur Funktion, genau wie bei `if` oder `for` die eingerückten Zeilen dazugehören.
- Für Funktionsnamen gelten dieselben Regeln wie für Variablen: klein, mit `_`, ohne Umlaute. Zwei leere Zeilen vor und nach jeder Funktion braucht Python nicht; sie machen das Programm aber übersichtlicher.

Beim Definieren passiert noch **nichts Sichtbares**. Python merkt sich nur: „Unter dem Namen `ansage` stehen diese Anweisungen." Deshalb erscheint in der Ausgabe zuerst `Programmstart` aus Zeile 7.

Erst ein **Aufruf** wie in Zeile 8 führt den Funktionskörper aus. Python springt in die Funktion, arbeitet sie von oben nach unten ab und macht danach in der Zeile unter dem Aufruf weiter. Zeile 9 ruft die Funktion ein zweites Mal auf, deshalb steht die Ansage zweimal im Terminal.

Erst definieren, dann aufrufen. Python liest das Programm von oben nach unten. Steht der Aufruf `ansage()` in Zeile 1, also vor dem `def`, kennt Python den Namen noch nicht:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ansage' isn't defined
```

Klammern nicht vergessen. Schreibst du nur `ansage` ohne `()`, gibt es keine Fehlermeldung, aber es passiert auch nichts: Die Funktion wird nicht aufgerufen.

11.2 Parameter: Funktionen Werte mitgeben

Die Ansage oben sagt immer dasselbe. Viel nützlicher ist sie, wenn du ihr beim Aufruf mitteilen kannst, um welchen Bus es geht:

k11_parameter.py

```
1 def ansage(linie, minuten):
2     print("Bus", linie, "fährt in", minuten, "Minuten.")
3
4
5 ansage(7, 5)
6 ansage(12, 3)
7 ansage(7, 20)
```

Terminal

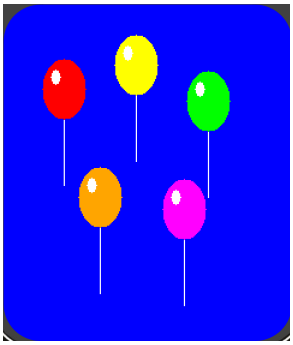
```
Bus 7 fährt in 5 Minuten.
Bus 12 fährt in 3 Minuten.
Bus 7 fährt in 20 Minuten.
```

In der Klammer hinter dem Funktionsnamen stehen jetzt zwei Namen, `linie` und `minuten`. Sie heißen **Parameter**. Ein Parameter ist eine Variable, die beim Aufruf ihren Wert bekommt. Beim Aufruf `ansage(7, 5)` wird `linie` zu `7` und `minuten` zu `5`. Die Werte beim Aufruf nennt man **Argumente**. Python ordnet sie der Reihe nach zu: das erste Argument dem ersten Parameter, das zweite dem zweiten.

Mit Parametern lohnen sich Funktionen besonders beim Zeichnen. Ein Ballon besteht aus drei Teilen. Mit einer Funktion zeichnest du ihn mit einer einzigen Zeile, an jede Stelle und in jeder Farbe:

k11_ballons.py

```
 1 import pbc
 2
 3
 4 def ballon(x, y, farbe):
 5     pbc.line(x, y + 25, x, y + 80, pbc.WHITE)
 6     pbc.ellipse(x, y, 18, 25, farbe, fill=True)
 7     pbc.ellipse(x - 7, y - 10, 4, 6, pbc.WHITE, fill=True)
 8
 9
10 pbc.clear(pbc.BLUE)
11 ballon(50, 70, pbc.RED)
12 ballon(110, 50, pbc.YELLOW)
13 ballon(170, 80, pbc.GREEN)
14 ballon(80, 160, pbc.ORANGE)
15 ballon(150, 170, pbc.MAGENTA)
```



- Zeile 4: Die Funktion `ballon` hat drei Parameter: die Mitte des Ballons (`x`, `y`) und seine `farbe`.
- Zeile 5 bis 7 zeichnen Schnur, Ballon und Glanzpunkt. Alle Positionen werden **aus `x` und `y` berechnet**: Die Schnur beginnt 25 Pixel unter der Mitte, der Glanzpunkt sitzt 7 Pixel links und 10 Pixel über der Mitte. Darum passen die Teile überall zusammen.
- Zeile 11 bis 15: fünf Aufrufe, fünf Ballons. Ohne Funktion wären das 15 Zeilen.

Gibst du beim Aufruf zu wenige oder zu viele Argumente an, bricht das Programm mit einem `TypeError` ab. Hier fehlt in Zeile 11 die Farbe:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 11, in <module>
TypeError: ballon() takes 3 positional arguments but 2 were given
```

`ballon()` erwartet 3 Werte (*positional arguments*), bekommen hat es 2.

Eine Funktion darf alles enthalten, was du bisher kennst, auch Schleifen und Wartebefehle. Diese Funktion zählt von einer Zahl herunter:

k11_countdown.py

```
 1 import pbc
 2 import time
 3
 4
 5 def countdown(von, farbe):
 6     for zahl in range(von, 0, -1):
 7         pbc.clear()
 8         pbc.text24(str(zahl), 112, 120, farbe)
 9         time.sleep(1)
10     pbc.clear()
11     pbc.text24("Los!", 95, 120, farbe)
12
13
14 countdown(3, pbc.YELLOW)
15 time.sleep(1)
16 countdown(5, pbc.GREEN)
```



Die `for`-Schleife in Zeile 6 bis 9 gehört zur Funktion, Zeile 10 und 11 ebenfalls (alle sind eingerückt). Zeile 14 zählt von 3 in Gelb herunter, Zeile 16 von 5 in Grün. Erst wenn die Funktion fertig ist, geht es mit der Zeile nach dem Aufruf weiter.

11.3 Standardwerte für Parameter

Manche Befehle kannst du mit oder ohne Angabe aufrufen: `pbc.clear()` färbt den Bildschirm schwarz, `pbc.clear(pbc.RED)` rot. `pbc.tone(440)` spielt einen Dauerton, `pbc.tone(440, 200)` einen kurzen Ton. Das geht auch bei eigenen Funktionen. Du gibst dem Parameter einen **Standardwert**:

k11_standard.py

```
 1 import pbc
 2 import time
 3
 4
 5 def piep(frequenz=880, dauer=150):
 6     print("Ton:", frequenz, "Hz,", dauer, "ms")
 7     pbc.tone(frequenz, dauer)
 8     time.sleep_ms(dauer + 50)
 9
10
11 piep()
12 piep(440)
13 piep(660, 400)
14 piep(dauer=600)
```

Terminal

```
Ton: 880 Hz, 150 ms
Ton: 440 Hz, 150 ms
Ton: 660 Hz, 400 ms
Ton: 880 Hz, 600 ms
```

- `frequenz=880` in Zeile 5 heißt: Gibt der Aufruf keine Frequenz an, nimmt die Funktion 880. Ebenso ist `dauer` ohne Angabe 150.
- Zeile 11: `piep()` ohne Argumente benutzt beide Standardwerte.
- Zeile 12: `piep(440)` ersetzt nur den ersten Wert, `dauer` bleibt 150.
- Zeile 14: Willst du nur die Dauer ändern, nennst du den Parameter beim Namen: `piep(dauer=600)`. So ein **Schlüsselwort-Argument** darf in beliebiger Reihenfolge stehen.
- Zeile 8: `pbc.tone` wartet nicht. Darum wartet die Funktion selbst etwas länger, als der Ton dauert, damit die Töne einzeln zu hören sind.

Parameter mit Standardwert müssen **hinter** den Parametern ohne Standardwert stehen. `def piep(frequenz=880, dauer):` ist verboten:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
SyntaxError: non-default argument follows default argument
```

Übersetzt: „Parameter ohne Standardwert folgt auf Parameter mit Standardwert.“ Bei einem `SyntaxError` läuft das Programm gar nicht erst los.

11.4 Programme in Funktionen aufteilen

Funktionen helfen nicht nur gegen Wiederholungen. Sie helfen auch, ein Programm zu **ordnen**: Jede Funktion erledigt eine Aufgabe und hat einen Namen, der diese Aufgabe beschreibt. Dieses Programm zeigt einen Wetterbericht für drei Tage:

k11_wetter.py

```
 1 import pbc
 2 import time
 3
 4
 5 def sonne(x, y):
 6     pbc.ellipse(x, y, 30, 30, pbc.YELLOW, fill=True)
 7
 8
 9 def wolke(x, y):
10     pbc.ellipse(x - 25, y + 5, 25, 18, pbc.WHITE, fill=True)
11     pbc.ellipse(x + 5, y - 8, 30, 24, pbc.WHITE, fill=True)
12     pbc.ellipse(x + 32, y + 6, 22, 16, pbc.WHITE, fill=True)
```

`sonne` und `wolke` zeichnen je ein Wettersymbol an die Stelle (`x`, `y`). Die Wolke besteht aus drei weißen Ellipsen, die sich überlappen.

k11_wetter.py

```
15 def zeige_tag(tag, temperatur, sonnig):
16     pbc.clear(pbc.BLUE)
17     pbc.text24(tag, 20, 20, pbc.WHITE)
18     if sonnig:
19         sonne(120, 130)
20     else:
21         wolke(115, 130)
22     pbc.text24(f"{temperatur} Grad", 65, 210, pbc.WHITE)
23     time.sleep(2)
24
25
26 zeige_tag("Montag", 21, True)
27 zeige_tag("Dienstag", 14, False)
28 zeige_tag("Mittwoch", 18, True)
```



Das Programm hat zwei Teile:

1. **Oben die Funktionen** (Zeile 5 bis 23). `zeige_tag` zeigt einen ganzen Tag: Hintergrund, Name, Symbol, Temperatur, dann 2 Sekunden Pause.
2. **Unten das Hauptprogramm** (Zeile 26 bis 28). Es ist nur drei Zeilen lang und liest sich fast wie ein Satz: „Zeige Montag, 21 Grad, sonnig."

`zeige_tag` ruft selbst wieder Funktionen auf (Zeile 19 und 21). Das ist ganz normal: Große Aufgaben werden in kleine zerlegt, kleine Aufgaben in noch kleinere. Willst du später ein Regensymbol, schreibst du eine Funktion `regen` und änderst nur `zeige_tag`. Das Hauptprogramm bleibt, wie es ist.

Gute Funktionen

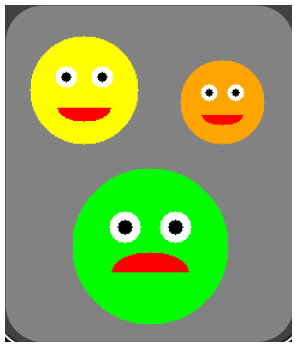
- Eine Funktion erledigt **eine** Aufgabe. Ihr Name sagt, was sie **tut** (`zeichne_ball`, `zeige_tag`, `piep`), oft ist es ein Verb.
- Alles, was sich von Aufruf zu Aufruf ändert, wird ein **Parameter**.
- Erst alle `def`, dann das Hauptprogramm. Die `import`-Zeilen stehen darüber ganz oben.

Ausprobieren: Das Gesicht in drei Größen

Dieses Programm zeichnet drei Gesichter mit einer einzigen Funktion `gesicht`. Groß oder klein, gelb oder grün, fröhlich oder traurig: Das entscheiden die Argumente.

k11_gesicht.py

```
 1 import pbc
 2
 3
 4 def auge(x, y, r):
 5     pbc.ellipse(x, y, r, r, pbc.WHITE, fill=True)
 6     pbc.ellipse(x, y, r // 2, r // 2, pbc.BLACK, fill=True)
 7
 8
 9 def mund(x, y, breite, farbe, froehlich):
10     hoehe = breite // 2
11     pbc.ellipse(x, y, breite, hoehe, pbc.RED, fill=True)
12     # eine Haelfte mit der Gesichtsfarbe uebermalen
13     if froehlich:
14         pbc.fill_rect(x - breite, y - hoehe, 2 * breite + 1, hoehe, farbe)
15     else:
16         pbc.fill_rect(x - breite, y + 1, 2 * breite + 1, hoehe, farbe)
17
18
19 def gesicht(x, y, r, farbe=pbc.YELLOW, froehlich=True):
20     pbc.ellipse(x, y, r, r, farbe, fill=True)
21     auge(x - r // 3, y - r // 4, r // 5)
22     auge(x + r // 3, y - r // 4, r // 5)
23     mund(x, y + r // 3, r // 2, farbe, froehlich)
24
25
26 pbc.clear(pbc.GRAY)
27 gesicht(65, 70, 45)
28 gesicht(180, 80, 35, pbc.ORANGE)
29 gesicht(120, 200, 65, pbc.GREEN, False)
```



- **`auge`** (Zeile 4 bis 6) zeichnet ein weißes Auge mit schwarzer Pupille. Die Pupille ist halb so groß wie das Auge (`r // 2`).
- **`mund`** (Zeile 9 bis 16) zeichnet eine rote Ellipse und übermalt dann eine Hälfte in der Gesichtsfarbe. Bleibt die untere Hälfte stehen, sieht der Mund fröhlich aus; bleibt die obere stehen, traurig.
- **`gesicht`** (Zeile 19 bis 23) ruft beide auf. Jede Größe hängt vom Radius `r` ab: Die Augen sitzen ein Drittel des Radius neben und ein Viertel über der Mitte, ihr Radius ist ein Fünftel von `r`. Darum stimmen die Verhältnisse bei jeder Größe.
- `farbe` und `froehlich` haben Standardwerte. Zeile 27 zeichnet also ein gelbes, fröhliches Gesicht, Zeile 29 ein grünes, trauriges.

Zum Verändern: Gib `gesicht` einen weiteren Parameter `nase` mit dem Standardwert `False`. Ist er `True`, bekommt das Gesicht eine kleine Nase. Zeichne damit fünf Gesichter in einer Reihe, vom kleinsten zum größten. Wie klein darf `r` werden, bis man das Gesicht nicht mehr erkennt?

Übungsaufgaben

Aufgabe 1 ⭐ Was gibt das Programm aus? Lies das Programm genau und schreibe auf, was im Terminal erscheint. Prüfe danach im Emulator.

k11_aufgabe1.py

```
def a():
    print("A")


def b(text):
    print("B", text)
    a()


print("Start")
b("eins")
a()
b("zwei")
print("Ende")
```

Aufgabe 2 ⭐ Finde die Fehler Das Programm soll drei Sterne auf blauen Hintergrund zeichnen, enthält aber drei Fehler. Teste, lies die Fehlermeldung, korrigiere und teste erneut. Notiere zu jedem Fehler die Fehlerart und die Zeile.

k11_aufgabe2.py

```
import pbc


def stern(x, y, farbe)
    pbc.line(x - 10, y, x + 10, y, farbe)
    pbc.line(x, y - 10, x, y + 10, farbe)
    pbc.line(x - 7, y - 7, x + 7, y + 7, farbe)
    pbc.line(x - 7, y + 7, x + 7, y - 7, farbe)


pbc.clear(pbc.BLUE)
stern(50, 60, pbc.YELLOW)
stern(120, 100)
Stern(190, 50, pbc.WHITE)
```

Aufgabe 3 ⭐⭐ Das Lauftagebuch Schreibe eine Funktion `balken(y, tag, km, farbe)`. Sie schreibt links den Tag (z. B. `"Mo"`), zeichnet daneben einen Balken, dessen Länge von `km` abhängt, und schreibt die Kilometer hinter den Balken. Zeige damit die gelaufenen Kilometer einer Woche untereinander an.

Aufgabe 4 ⭐⭐ Ein besserer Countdown Ändere `k11_countdown.py`:

- `farbe` bekommt den Standardwert `pbc.WHITE`.
- Ein neuer Parameter `ziel` mit dem Standardwert `"Los!"` bestimmt, was am Ende erscheint.
- Bei jeder Zahl erklingt ein kurzer Ton, am Ende ein hoher, langer Ton.

Rufe die Funktion einmal nur mit `3` auf und einmal mit `5`, Rot und dem Text `"Start!"`.

Aufgabe 5 ⭐⭐⭐ Flaggen Viele Flaggen bestehen aus drei gleich breiten Streifen, entweder übereinander (Deutschland, Österreich) oder nebeneinander (Frankreich, Italien). Schreibe eine Funktion `flagge(x, y, f1, f2, f3, quer=True)`, die beide Sorten zeichnen kann. Schreibe eine zweite Funktion, die eine Flagge zeichnet und ihren Namen darunter schreibt. Zeige sechs Flaggen auf einem Bildschirm.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `def name(parameter):` definiert eine Funktion. Die eingerückten Zeilen darunter sind ihr Körper.
- Die Definition allein tut nichts. Erst der Aufruf `name(argumente)` führt den Körper aus. Die Definition muss **vor** dem ersten Aufruf stehen.
- **Parameter** sind Variablen der Funktion, die beim Aufruf ihre Werte bekommen. Die Werte beim Aufruf heißen **Argumente**; sie werden der Reihe nach zugeordnet.
- Falsche Anzahl von Argumenten: `TypeError`.
- Standardwerte: `def piep(frequenz=880):` – das Argument darf dann fehlen. Mit `piep(dauer=600)` gibst du gezielt einen Parameter an.
- Funktionen dürfen zeichnen, warten und andere Funktionen aufrufen.
- Gute Ordnung: `import`, dann alle Funktionen, dann das kurze Hauptprogramm.

KAPITEL 12

Funktionen mit Rückgabewert

Worum geht's? Die Funktionen aus Kapitel 11 *tun* etwas: Sie zeichnen, schreiben oder spielen Töne. Viele Funktionen *berechnen* aber etwas und geben das Ergebnis zurück, so wie `len(text)` eine Zahl liefert. In diesem Kapitel schreibst du solche Funktionen, lernst, wo ihre Variablen gelten, und prüfst mit kleinen Tests, ob sie richtig rechnen.

12.1 `return` – die Funktion liefert ein Ergebnis

k12_fahrenheit.py

```
1 def fahrenheit(celsius):
2     return celsius * 9 / 5 + 32
3
4
5 print(fahrenheit(20))
6 sommer = fahrenheit(25)
7 print("Sommertag:", sommer, "°F")
8 for c in range(0, 41, 10):
9     print(f"{c} °C = {fahrenheit(c)} °F")
10 if fahrenheit(30) > 85:
11     print("30 °C ist heiß!")
```

Terminal

```
68.0
Sommertag: 77.0 °F
0 °C = 32.0 °F
10 °C = 50.0 °F
20 °C = 68.0 °F
30 °C = 86.0 °F
40 °C = 104.0 °F
30 °C ist heiß!
```

In Zeile 2 steht `return`. Das Wort heißt „zurückgeben". Die Funktion rechnet den Ausdruck dahinter aus und gibt das Ergebnis an die Stelle zurück, an der sie aufgerufen wurde. Dieses Ergebnis heißt **Rückgabewert**.

Ein Aufruf wie `fahrenheit(20)` steht damit für einen Wert, hier `68.0`. Du kannst ihn überall dort benutzen, wo ein Wert stehen darf:

- in `print` ausgeben (Zeile 5),
- in einer Variablen speichern (Zeile 6),
- in einem f-String einsetzen (Zeile 9),
- in einem Vergleich prüfen (Zeile 10).

`return` beendet die Funktion **sofort**. Das nutzt diese Funktion aus. Sie berechnet den Eintrittspreis für ein Schwimmbad:

k12_eintritt.py

```
1  def eintritt(alter):
2      if alter < 6:
3          return 0
4      if alter < 18:
5          return 4
6      if alter >= 65:
7          return 6
8      return 9
9
10
11 print("Kind (4):", eintritt(4))
12 print("Schülerin (14):", eintritt(14))
13 zusammen = eintritt(41) + eintritt(5)
14 print("Vater mit Kind:", zusammen)
```

Terminal

```
Kind (4): 0
Schülerin (14): 4
Vater mit Kind: 9
```

Für ein Kind mit 4 Jahren ist schon in Zeile 2 die Bedingung wahr. Zeile 3 gibt `0` zurück, und die restlichen Zeilen der Funktion laufen gar nicht mehr. Deshalb braucht es hier kein `elif`: Wer bis Zeile 8 kommt, ist mindestens 18 und jünger als 65. In Zeile 13 rechnet das Programm mit zwei Rückgabewerten. Das geht, weil jeder Aufruf für eine Zahl steht.

`print` ist nicht `return`. In `k12_fehler_print.py` steht in der Funktion `eintritt` statt `return 4` nur `print(4)` und statt `return 9` nur `print(9)`. Die Funktion *zeigt* den Preis also an, gibt ihn aber nicht zurück. Zeile 8 lautet `summe = eintritt(41) + eintritt(12)`:

Terminal

```
9
4

Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
TypeError: unsupported types for __add__: 'NoneType', 'NoneType'
```

`9` und `4` erscheinen im Terminal, aber Zeile 8 kann damit nicht rechnen. Eine Funktion ohne `return` gibt den besonderen Wert **`None`** zurück („nichts“). `None + None` geht nicht, daher der `TypeError`. Faustregel: Eine Funktion, die etwas *berechnet*, gibt das Ergebnis mit `return` zurück. Was angezeigt wird, entscheidet das Hauptprogramm.

12.2 Funktionen, die `True` oder `False` liefern

Ein Rückgabewert kann auch ein Wahrheitswert sein. Solche Funktionen beantworten eine Ja-Nein-Frage. Man gibt ihnen gern Namen wie `ist_…` oder `hat_…`, dann lesen sich Bedingungen wie ein Satz:

k12_schaltjahr.py

```
 1 def ist_schaltjahr(jahr):
 2     if jahr % 400 == 0:
 3         return True
 4     if jahr % 100 == 0:
 5         return False
 6     return jahr % 4 == 0
 7
 8
 9 print(ist_schaltjahr(2024), ist_schaltjahr(2025))
10 for jahr in range(2020, 2041):
11     if ist_schaltjahr(jahr):
12         print(jahr, "hat 366 Tage")
```

Terminal

```
True False
2020 hat 366 Tage
2024 hat 366 Tage
2028 hat 366 Tage
2032 hat 366 Tage
2036 hat 366 Tage
2040 hat 366 Tage
```

Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, außer es ist durch 100 teilbar. Durch 400 teilbare Jahre sind aber doch wieder Schaltjahre. Die Funktion prüft diese Regeln von der strengsten zur allgemeinsten. Zeile 6 gibt direkt das Ergebnis des Vergleichs zurück: `jahr % 4 == 0` ist schon `True` oder `False`. Ein `if … return True else return False` wäre unnötig lang.

In Zeile 11 steht `if ist_schaltjahr(jahr):`. Ein `== True` ist nicht nötig, denn die Funktion liefert schon den Wahrheitswert.

Im Spiel beantworten solche Funktionen Fragen wie „Ist die Figur im Ziel?“. Steuere den gelben Punkt mit dem Joystick nach rechts und nach unten in das grüne Quadrat:

k12_ziel.py

```
 1 import pbc
 2 import time
 3
 4 ZIEL_X = 150
 5 ZIEL_Y = 180
 6
 7
 8 def zwischen(wert, kleinster, groesster):
 9     return wert >= kleinster and wert <= groesster
10
11
12 def im_ziel(x, y):
13     return zwischen(x, ZIEL_X, ZIEL_X + 40) and zwischen(y, ZIEL_Y, ZIEL_Y + 40)
```

`zwischen` prüft, ob ein Wert zwischen zwei Grenzen liegt. `im_ziel` benutzt `zwischen` zweimal, für x und für y. Eine Funktion mit Rückgabewert darf also andere Funktionen mit Rückgabewert aufrufen.

k12_ziel.py

```
16 x = 30
17 y = 40
18 pbc.clear()
19 pbc.rect(ZIEL_X, ZIEL_Y, 46, 46, pbc.GREEN)
20 while not im_ziel(x, y):
21     pbc.fill_rect(x, y, 6, 6, pbc.BLACK)
22     if pbc.pressed_right():
23         x += 3
24     if pbc.pressed_down():
25         y += 3
26     pbc.fill_rect(x, y, 6, 6, pbc.YELLOW)
27     time.sleep_ms(20)
28 pbc.text24("Im Ziel!", 70, 100, pbc.GREEN)
```



- Zeile 20: Die Schleife läuft, solange der Punkt **nicht** im Ziel ist. Sie liest sich fast wie Deutsch.
- `ZIEL_X` und `ZIEL_Y` sind Konstanten (große Buchstaben). Willst du das Ziel verschieben, änderst du nur Zeile 4 und 5.

12.3 Mehrere Rückgabewerte

Manchmal hat eine Rechnung zwei Ergebnisse. Eine Laufzeit von 754 Sekunden sind 12 Minuten und 34 Sekunden. Hinter `return` dürfen deshalb mehrere Werte stehen, durch Kommas getrennt:

k12_zeit.py

```
1 def minuten_sekunden(sekunden):
2     return sekunden // 60, sekunden % 60
3
4
5 m, s = minuten_sekunden(754)
6 print("Laufzeit:", m, "min", s, "s")
7 print(minuten_sekunden(125))
```

Terminal

```
Laufzeit: 12 min 34 s
(2, 5)
```

In Zeile 5 stehen links vom `=` zwei Variablen. Python verteilt die beiden Rückgabewerte der Reihe nach: Der erste kommt in `m`, der zweite in `s`. Das nennt man **Entpacken**.

Zeile 7 gibt den Rückgabewert direkt aus. Du siehst `(2, 5)` in runden Klammern. Mehrere Werte, mit Kommas zusammengefasst, bilden ein **Tupel**. Für dich reicht im Moment: Ein Tupel hält mehrere Werte zusammen, und beim Entpacken müssen links genauso viele Variablen stehen, wie Werte kommen.

Stimmt die Anzahl nicht, gibt es einen `ValueError`. Hier sollen zwei Werte auf drei Variablen `h, m, s` verteilt werden:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
ValueError: need more than 2 values to unpack
```

Mehrere Rückgabewerte sind auch praktisch für Positionen: Eine Funktion `zufallsort()` könnte `return x, y` liefern, und das Hauptprogramm holt sich mit `x, y = zufallsort()` einen neuen Ort.

12.4 Lokale und globale Variablen, `global`

Variablen, die **innerhalb** einer Funktion entstehen, gibt es nur dort. Sie heißen **lokale Variablen**. Auch Parameter sind lokal. Sobald die Funktion fertig ist, sind sie verschwunden:

k12_lokal.py

```
1 def flaeche(breite, hoehe):
2     ergebnis = breite * hoehe
3     return ergebnis
4
5
6 print(flaeche(3, 4))
7 print(ergebnis)
```

Terminal

```
12

Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
NameError: name 'ergebnis' isn't defined
```

`ergebnis` lebt nur in der Funktion `flaeche`. Draußen in Zeile 7 kennt Python den Namen nicht. Wer den Wert draußen braucht, bekommt ihn über `return` (Zeile 3).

Variablen, die im Hauptprogramm entstehen, heißen **globale Variablen**. Eine Funktion darf sie **lesen**. Weist eine Funktion einem Namen aber einen Wert zu, entsteht eine **neue lokale** Variable, auch wenn es draußen schon eine mit diesem Namen gibt. Das ist gewollt: Eine Funktion kann so nicht aus Versehen Variablen des Hauptprogramms überschreiben. Willst du eine globale Variable in einer Funktion **verändern**, gibt es deshalb einen Fehler:

k12_global_fehler.py

```
1 punkte = 0
2
3
4 def treffer(wert):
5     punkte = punkte + wert
6
7
8 treffer(10)
9 print(punkte)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
  File "<stdin>", line 5, in treffer
NameError: local variable referenced before assignment
```

Weil in Zeile 5 `punkte = …` steht, ist `punkte` in der ganzen Funktion lokal. Rechts vom `=` soll diese lokale Variable gelesen werden, sie hat aber noch keinen Wert („referenced before assignment“). Beachte den Traceback: Er hat **zwei Zeilen** mit `File`. Die obere zeigt den Aufruf in Zeile 8, die untere die Zeile 5 in der Funktion `treffer`, in der es wirklich knallt. Lies einen Traceback mit mehreren Stufen also wieder von unten: Die unterste `File`-Zeile ist die Fehlerstelle, die darüber zeigen den Weg dorthin.

Die Lösung ist die Anweisung `global`. Sie sagt der Funktion: „Mit `punkte` ist die globale Variable gemeint.“ So funktioniert `treffer` (`k12_global.py`):

```
def treffer(wert):
    global punkte
    punkte = punkte + wert
```

Eine Funktion, die `punkte` nur liest (z. B. mit `print`), braucht kein `global`.

`global` ist bequem, macht größere Programme aber unübersichtlich: Jede Funktion könnte jede Variable verändern. Oft geht es ohne. Gib den Wert hinein und hol das Ergebnis mit `return` heraus (`k12_ohne_global.py`):

```
def mit_treffer(punkte, wert):
    return punkte + wert


punkte = mit_treffer(punkte, 10)
```

So sieht man im Hauptprogramm genau, wo sich `punkte` ändert.

12.5 Funktionen testen

Eine Funktion mit Rückgabewert lässt sich gut **testen**: Du rufst sie mit Werten auf, deren richtiges Ergebnis du kennst, und vergleichst. Am einfachsten geht das mit `print`. Noch bequemer ist die Anweisung `assert` („versichern"). Sie prüft eine Bedingung. Ist sie wahr, passiert nichts. Ist sie falsch, bricht das Programm mit einem `AssertionError` ab.

k12_testen.py

```
 1 def ist_schaltjahr(jahr):
 2     if jahr % 400 == 0:
 3         return True
 4     if jahr % 100 == 0:
 5         return False
 6     return jahr % 4 == 0
 7
 8
 9 # Tests mit print
10 print(ist_schaltjahr(2024), "erwartet: True")
11 print(ist_schaltjahr(1900), "erwartet: False")
12
13 # Tests mit assert
14 assert ist_schaltjahr(2024)
15 assert not ist_schaltjahr(2023)
16 assert ist_schaltjahr(2000)
17 assert not ist_schaltjahr(1900)
18 print("Alle Tests bestanden")
```

Terminal

```
True erwartet: True
False erwartet: False
Alle Tests bestanden
```

- Zeile 10 und 11: Bei `print`-Tests musst du selbst hinschauen, ob Ergebnis und Erwartung übereinstimmen.
- Zeile 14 bis 17: Die `assert`-Tests sind still, solange alles stimmt. Erst wenn alle bestanden sind, erscheint Zeile 18.

- Gute Tests prüfen **typische Fälle** (2024, 2023) und **Grenzfälle** (1900, 2000). Bei Grenzfällen passieren die meisten Fehler.

Und so sieht es aus, wenn ein Test scheitert. Diese Funktion hat die Regel mit den Jahrhunderten vergessen:

k12_test_rot.py

```
1 def ist_schaltjahr(jahr):
2     return jahr % 4 == 0
3
4
5 assert ist_schaltjahr(2024), "2024 ist ein Schaltjahr"
6 assert not ist_schaltjahr(2023), "2023 ist kein Schaltjahr"
7 assert not ist_schaltjahr(1900), "1900 ist kein Schaltjahr"
8 assert ist_schaltjahr(2000), "2000 ist ein Schaltjahr"
9 print("Alle Tests bestanden")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
AssertionError: 1900 ist kein Schaltjahr
```

Die Tests für 2024 und 2023 sind bestanden, der für 1900 nicht. Hinter dem Komma darf bei `assert` ein Text stehen; er erscheint in der Fehlermeldung. Ohne Text steht hinter `AssertionError:` nichts, dann verrät nur die Zeilennummer, welcher Test gescheitert ist. Ein Hinweisfenster wie bei anderen Fehlern öffnet PBC+ Studio bei einem `AssertionError` nicht.

> Schreib die Tests unter die Funktion, am besten **bevor** du sie programmierst. Sie laufen bei jedem Start mit: Machst du später etwas kaputt, merkst du es sofort.

Ausprobieren: Die Notenrechner-Werkzeugkiste

Dieses Programm fragt nach den Punkten mehrerer Klassenarbeiten, berechnet für jede die Note und zum Schluss den Durchschnitt. Es besteht aus kleinen Werkzeugen, die jeweils eine Frage beantworten:

k12_notenrechner.py

```
1 def note(punkte, maximal):
2     prozent = punkte * 100 / maximal
3     if prozent >= 92:
4         return 1
5     if prozent >= 81:
6         return 2
7     if prozent >= 67:
8         return 3
9     if prozent >= 50:
10        return 4
```

```
    if prozent >= 30:
        return 5
    return 6


def ist_bestanden(n):
    return n <= 4


def frage_zahl(frage):
    return int(input(frage))
```

- `note` rechnet die Punkte in Prozent um und gibt die passende Note zurück. Jedes `return` beendet die Funktion, deshalb reichen einfache `if`-Abfragen von oben nach unten.
- `ist_bestanden` beantwortet eine Ja-Nein-Frage.
- `frage_zahl` fasst `input` und `int` zusammen und gibt die Zahl zurück.

k12_notenrechner.py

```
# Tests: Stimmen die Werkzeuge?
assert note(46, 50) == 1 and note(45, 50) == 2
assert note(25, 50) == 4 and note(24, 50) == 5
assert ist_bestanden(4) and not ist_bestanden(4.1)

maximal = frage_zahl("Höchstpunktzahl? ")
anzahl = frage_zahl("Wie viele Arbeiten? ")
summe = 0
for i in range(anzahl):
    n = note(frage_zahl(f"Punkte in Arbeit {i + 1}? "), maximal)
    print("Note:", n)
    summe += n
schnitt = round(summe / anzahl, 1)
print("Durchschnitt:", schnitt)
if ist_bestanden(schnitt):
    print("Bestanden!")
```

Terminal

```
Höchstpunktzahl? 50
Wie viele Arbeiten? 3
Punkte in Arbeit 1? 47
Note: 1
Punkte in Arbeit 2? 38
Note: 3
Punkte in Arbeit 3? 41
Note: 2
Durchschnitt: 2.0
Bestanden!
```

- Zeile 25 bis 27 testen die Werkzeuge bei jedem Start an den Grenzen: 46 von 50 Punkten sind genau 92 %, 25 Punkte genau 50 %.
- Zeile 33 setzt einen Rückgabewert direkt als Argument ein: Erst liefert `frage_zahl` die Punkte, dann rechnet `note` daraus die Note.

- Zeile 36: `round(zahl, 1)` rundet auf eine Stelle nach dem Komma.

Zum Verändern: Schreibe eine Funktion `note_als_wort(n)`, die für 1 `"sehr gut"`, für 2 `"gut"` usw. zurückgibt, und gib das Wort zu jeder Note mit aus. Ergänze Tests dafür. Was passiert, wenn du bei „Wie viele Arbeiten?“ eine 0 eingibst? Wie kannst du das verhindern?

Übungsaufgaben

Aufgabe 1 ⭐ Wie schnell? Schreibe eine Funktion `kmh(meter, sekunden)`, die die Geschwindigkeit in km/h zurückgibt (Meter pro Sekunde mal 3,6). Teste sie mit `assert`: 100 m in 10 s sind 36 km/h, 1000 m in 3600 s sind 1 km/h. Wie schnell ist ein 100-m-Lauf in 12,5 s? Wie schnell war der Marathon-Weltrekord (42 195 m in 2 h 1 min 9 s)?

Aufgabe 2 ⭐ Was gibt das Programm aus? Schreibe auf, was im Terminal erscheint. Prüfe danach im Emulator.

k12_aufgabe2.py

```
1 def dreifach(x):
2     return 3 * x
3
4
5 def zeige(x):
6     print("Wert:", x)
7
8
9 a = dreifach(4)
10 b = dreifach(a) - 10
11 zeige(b)
12 c = zeige(5)
13 print(c)
14 print(dreifach(dreifach(1)))
```

Aufgabe 3 ⭐⭐ Finde die Fehler Nach drei verlorenen Leben soll „Game over“ erscheinen. Das Programm hat zwei Fehler. Nur einer davon erzeugt eine Fehlermeldung. Finde beide.

k12_aufgabe3.py

```
1 leben = 3
2
3
4 def verliere_leben():
5     leben = leben - 1
```

```
 8 def ist_vorbei():
 9     if leben == 0:
10         print(True)
11     else:
12         print(False)
13
14
15 verliere_leben()
16 verliere_leben()
17 verliere_leben()
18 if ist_vorbei():
19     print("Game over")
```

Aufgabe 4 ⭐⭐ Vokale zählen Schreibe eine Funktion `ist_vokal(zeichen)`, die `True` liefert, wenn das Zeichen ein Vokal ist (auch Umlaute, groß und klein). Schreibe damit eine Funktion `zaehle_vokale(text)`. Teste beide mit `assert`, auch mit dem leeren Text `""`.

Aufgabe 5 ⭐⭐ Erst die Tests, dann die Funktion Öffne `k12_aufgabe5.py`. Die Tests sind schon fertig, die Funktion `quersumme` aber nicht. Schreibe sie so, dass alle Tests bestehen. Die Quersumme von 123 ist 1 + 2 + 3 = 6. Tipp: `zahl % 10` ist die letzte Ziffer, `zahl // 10` schneidet sie ab.

Aufgabe 6 ⭐⭐⭐ Pasch! Schreibe eine Funktion `wuerfle_zwei()`, die zwei Würfelzahlen zurückgibt, und eine Funktion `ist_pasch(a, b)`. Das Hauptprogramm würfelt so lange, bis ein Pasch fällt, gibt jeden Wurf aus und am Ende die Anzahl der Würfe.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `return wert` beendet die Funktion und gibt `wert` an die Aufrufstelle zurück. Der Aufruf steht dann für diesen **Rückgabewert**.
- Ohne `return` liefert eine Funktion `None`. `print` zeigt etwas an, `return` gibt es zurück.
- Funktionen wie `ist_schaltjahr` liefern `True` oder `False` und passen direkt hinter `if` und `while`.
- `return a, b` gibt mehrere Werte zurück (ein Tupel); `x, y = f()` entpackt sie.
- Variablen in einer Funktion sind **lokal**. Globale Variablen darf eine Funktion lesen; zum Verändern braucht sie `global name`. Besser: Werte als Parameter hinein, Ergebnis mit `return` heraus.
- `assert bedingung, "Text"` testet: still, wenn alles stimmt, sonst `AssertionError`. Typische Fälle und Grenzfälle testen.

KAPITEL 13

Listen

Worum geht's? Bisher hatte jede Variable genau einen Wert. Für fünf Weitsprünge, sieben Wochentage oder eine Bestenliste bräuchtest du dann fünf, sieben oder zehn Variablen. Eine **Liste** speichert beliebig viele Werte unter einem einzigen Namen. Du lernst, Listen anzulegen, zu verändern, zu durchlaufen und auszuwerten.

13.1 Listen anlegen und auf Elemente zugreifen

Eine Liste schreibst du in **eckige Klammern** `[ ]`. Die Werte darin trennst du durch Kommas. Jeder Wert in der Liste heißt **Element**.

k13_wochentage.py

```
1 # Eine Liste mit den Tagen der Woche
2 tage = ["Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"]
3
4 print(tage)
5 print(tage[0])
6 print(tage[4])
7 print(tage[-1])
8 print(f"Die Woche hat {len(tage)} Tage.")
9 print(tage[1:3])
```

Terminal

```
['Mo', 'Di', 'Mi', 'Do', 'Fr', 'Sa', 'So']
Mo
Fr
So
Die Woche hat 7 Tage.
['Di', 'Mi']
```

- Zeile 2 legt eine Liste mit sieben Texten an und gibt ihr den Namen `tage`. Eine Liste ist ein eigener Datentyp, so wie Zahlen und Texte.
- Zeile 4 gibt die ganze Liste aus, mit Klammern und Anführungszeichen.
- Zeile 5 holt ein einzelnes Element. Die Zahl in den eckigen Klammern heißt **Index**. Wie bei den Zeichen eines Textes (Kapitel 10) beginnt die Zählung bei **0**: `tage[0]` ist `"Mo"`, `tage[4]` ist `"Fr"`.
- Zeile 7: Ein negativer Index zählt von hinten. `tage[-1]` ist das letzte Element, `tage[-2]` das vorletzte.
- Zeile 8: `len(tage)` liefert die Anzahl der Elemente, hier 7.
- Zeile 9: Auch Teilstücke gehen wie bei Texten. `tage[1:3]` ist eine neue Liste mit den Elementen 1 und 2. Das Element mit Index 3 gehört nicht mehr dazu.

Eine Liste kann Zahlen, Texte oder beides enthalten: `[3, 1, 4]`, `["Ja", "Nein"]`, `[12, "Mai", 2.5]`. Die **leere Liste** `[]` hat gar kein Element. Sie ist oft der Anfang einer Liste, die das Programm erst füllt.

> **Den Index gibt es nicht.** Eine Liste mit 7 Elementen hat die Indizes 0 bis 6. Das Programm `k13_fehler_index.py` gibt erst `tage[6]` aus (Zeile 3), dann `tage[7]` (Zeile 4):
>
> Terminal
> ```
> So
>
> Traceback (most recent call last):
>   File "<stdin>", line 4, in <module>
> IndexError: list index out of range
> ```
>
> Zeile 3 klappt noch („So“), Zeile 4 bricht mit `IndexError` ab. Der höchste erlaubte Index ist immer `len(liste) - 1`.

13.2 Ändern, anhängen, entfernen: `append`, `remove`, `pop`

Texte kann man nicht verändern, Listen schon. Eine Playlist wird beim Hören umgestellt, ergänzt und gekürzt:

k13_playlist.py
```
 1 playlist = ["Intro", "Sommer", "Regen"]
 2 print(playlist)
 3
 4 playlist[1] = "Winter"
 5 playlist.append("Finale")
 6 playlist.insert(0, "Start")
 7 print(playlist)
 8
 9 playlist.remove("Regen")
10 print(playlist)
11
12 letztes = playlist.pop()
13 print("Herausgenommen:", letztes)
14 erstes = playlist.pop(0)
15 print("Herausgenommen:", erstes)
16 print(playlist)
```

Terminal
```
['Intro', 'Sommer', 'Regen']
['Start', 'Intro', 'Winter', 'Regen', 'Finale']
['Start', 'Intro', 'Winter', 'Finale']
Herausgenommen: Finale
Herausgenommen: Start
['Intro', 'Winter']
```

- Zeile 4 **ersetzt** ein Element: Links vom `=` steht die Liste mit Index, rechts der neue Wert. Aus „Sommer“ wird „Winter“.

- `append` hängt ein Element **hinten** an (Zeile 5). Die Liste wird um eins länger.
- `insert(stelle, wert)` fügt an einer bestimmten Stelle ein (Zeile 6). Die Elemente dahinter rücken eins nach hinten.
- `remove(wert)` sucht den Wert und entfernt ihn (Zeile 9). Kommt er mehrmals vor, verschwindet nur das erste Vorkommen.
- `pop()` nimmt das **letzte** Element heraus und liefert es zurück (Zeile 12). Du kannst es also in einer Variablen speichern. `pop(0)` nimmt das Element mit Index 0 heraus (Zeile 14).

`append`, `insert`, `remove` und `pop` heißen **Methoden**: Befehle, die zu einem bestimmten Wert gehören. Man schreibt sie mit Punkt hinter den Namen der Liste, so wie `upper()` hinter einen Text. Anders als `upper()` liefern sie aber keine neue Liste, sondern verändern die Liste selbst.

> **`remove` braucht einen Wert, den es gibt.** Steht der Wert nicht in der Liste, bricht das Programm ab. Hier wurde `playlist.remove("Sonne")` in Zeile 3 aufgerufen:
>
> Terminal
> ```
> Traceback (most recent call last):
>   File "<stdin>", line 3, in <module>
> ValueError: object not in sequence
> ```
>
> „object not in sequence“ heißt: Das Objekt steht nicht in der Folge. Auch das Hinweisfenster von PBC+ Studio sagt, dass das gesuchte Element nicht in der Liste ist. Abhilfe: vorher mit `in` prüfen (Abschnitt 13.4), z. B. `if "Sonne" in playlist:`. Ebenso bricht `pop()` bei einer leeren Liste mit `IndexError: pop from empty list` ab.

13.3 Listen durchlaufen mit `for`

Mit `for` hast du schon Zahlen von `range` und die Zeichen eines Textes durchlaufen. Genauso geht es mit einer Liste:

k13_einkauf.py
```
 1 einkauf = ["Brot", "Milch", "Aepfel", "Kaese"]
 2
 3 # Jedes Element der Reihe nach
 4 for artikel in einkauf:
 5     print("-", artikel)
 6
 7 # Mit der Nummer: i laeuft von 0 bis len - 1
 8 for i in range(len(einkauf)):
 9     print(f"{i + 1}. {einkauf[i]}")
10
11 # Eine neue Liste Schritt fuer Schritt fuellen
12 quadrate = []
13 for zahl in range(1, 6):
14     quadrate.append(zahl * zahl)
15 print(quadrate)
```

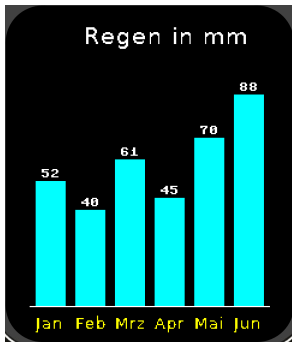
Terminal
```
- Brot
- Milch
- Aepfel
- Kaese
1. Brot
2. Milch
3. Aepfel
4. Kaese
[1, 4, 9, 16, 25]
```

- Zeile 4: Die Variable `artikel` bekommt bei jedem Durchlauf das nächste Element: erst „Brot“, dann „Milch“ und so weiter. Diese Form ist am kürzesten, wenn du nur die Werte brauchst.
- Zeile 8: Brauchst du auch die **Position**, lässt du einen Index laufen. `range(len(einkauf))` liefert 0, 1, 2, 3 – genau die gültigen Indizes. `einkauf[i]` holt dann das Element.
- Zeile 12–15: Ein häufiges Muster. Du beginnst mit einer leeren Liste und hängst in der Schleife mit `append` ein Element nach dem anderen an.

Listen und Schleifen zusammen sind ideal für Diagramme. Zwei Listen gehören hier zusammen: `monate[i]` ist der Name zum Wert `regen[i]`.

k13_regen.py

```
 1 import pbc
 2
 3 # Regenmenge in mm, Januar bis Juni
 4 monate = ["Jan", "Feb", "Mrz", "Apr", "Mai", "Jun"]
 5 regen = [52, 40, 61, 45, 70, 88]
 6
 7 pbc.clear()
 8 pbc.text18("Regen in mm", 65, 15, pbc.WHITE)
 9 pbc.hline(20, 250, 200, pbc.WHITE)
10
11 for i in range(len(regen)):
12     x = 25 + i * 33
13     hoehe = regen[i] * 2
14     pbc.fill_rect(x, 250 - hoehe, 25, hoehe, pbc.CYAN)
15     pbc.text(str(regen[i]), x + 4, 240 - hoehe, pbc.WHITE)
16     pbc.text12(monate[i], x, 256, pbc.YELLOW)
```



- Zeile 11: `i` läuft über alle Indizes der Liste `regen`.
- Zeile 12: Jeder Balken steht 33 Pixel weiter rechts als der vorige.
- Zeile 13: Ein Millimeter Regen wird 2 Pixel hoch gezeichnet.
- Zeile 14: Der Balken soll auf der Grundlinie bei y = 250 stehen. Seine linke obere Ecke liegt deshalb bei `250 - hoehe`.
- Zeile 15 und 16 beschriften den Balken mit dem Wert und dem Monatsnamen aus der zweiten Liste.

13.4 Auswerten: `len`, `sum`, `min`, `max`, `sorted`, `in`

Für die häufigsten Fragen an eine Liste hat Python fertige Funktionen:

k13_weitsprung.py

```
 1 # Weitsprung: fuenf Versuche in Metern
 2 weiten = [4.12, 3.85, 4.40, 3.97, 4.25]
 3
 4 print("Versuche:", len(weiten))
 5 print("Bester:", max(weiten))
 6 print("Schlechtester:", min(weiten))
 7 print("Summe:", sum(weiten))
 8 schnitt = sum(weiten) / len(weiten)
 9 print("Durchschnitt:", round(schnitt, 2))
10
11 print(sorted(weiten))
12 print(sorted(weiten, reverse=True))
13 print(weiten)
14
15 print(4.40 in weiten)
16 print(5.00 in weiten)
```

Terminal

```
Versuche: 5
Bester: 4.4
Schlechtester: 3.85
Summe: 20.59
Durchschnitt: 4.12
[3.85, 3.97, 4.12, 4.25, 4.4]
[4.4, 4.25, 4.12, 3.97, 3.85]
[4.12, 3.85, 4.4, 3.97, 4.25]
True
False
```

`len` zählt die Elemente, `sum` addiert sie (nur bei Zahlen), `min` und `max` liefern das kleinste und das größte Element.

- Zeile 8: Den Durchschnitt gibt es nicht als fertige Funktion. Er ist die Summe geteilt durch die Anzahl.
- Zeile 11 und 12: `sorted` liefert eine **neue**, sortierte Liste, mit `reverse=True` absteigend. Die ursprüngliche Liste bleibt unverändert (Zeile 13).
- Zeile 15 und 16: `in` kennst du schon von Texten. Bei Listen fragt es, ob ein Element mit genau diesem Wert vorkommt.

Zwei weitere Methoden helfen beim Suchen: `liste.count(wert)` zählt, wie oft ein Wert vorkommt, und `liste.index(wert)` liefert den Index seines ersten Vorkommens.

Auch Texte lassen sich sortieren:

k13_sortieren.py

```
1 namen = ["Mia", "Ben", "Alexander", "Zoe", "Lea"]
2 print(sorted(namen))
3 print(sorted(namen, key=len))
4
5 namen.sort()
6 print(namen)
7
8 gemischt = ["Bea", "ali", "Cem"]
9 print(sorted(gemischt))
```

Terminal

```
['Alexander', 'Ben', 'Lea', 'Mia', 'Zoe']
['Lea', 'Zoe', 'Mia', 'Ben', 'Alexander']
['Alexander', 'Ben', 'Lea', 'Mia', 'Zoe']
['Bea', 'Cem', 'ali']
```

- Zeile 2 sortiert alphabetisch.
- Zeile 3: Mit `key=len` sortiert `sorted` nicht nach den Texten selbst, sondern nach dem, was `len` für jeden Text liefert, also nach der Länge. Hinter `key=` steht der Name einer Funktion, **ohne** Klammern. Bei gleicher Länge bleibt die ursprüngliche Reihenfolge in MicroPython nicht unbedingt erhalten: „Mia“, „Ben“, „Zoe“ und „Lea“ sind alle vier drei Zeichen lang und stehen danach anders als vorher.
- Zeile 5: Die **Methode** `sort()` sortiert die Liste selbst um und liefert nichts zurück. `sorted(...)` dagegen liefert eine neue Liste.
- Zeile 9: Großbuchstaben kommen beim Sortieren **vor** allen Kleinbuchstaben. Deshalb steht „ali“ hinter „Cem“. Der Grund sind die Zeichencodes aus Kapitel 10: `ord("C")` ist 67, `ord("a")` ist 97.

> `werte = werte.sort()` ist ein typischer Fehler: `sort()` liefert `None` („nichts“), und danach ist die Liste weg. Schreibe entweder `werte.sort()` oder `werte = sorted(werte)`. Und `max` oder `min` einer leeren Liste bricht ab mit `ValueError: arg is an empty sequence`.

13.5 Listen und Funktionen

Eine Liste kann wie jeder andere Wert an eine Funktion übergeben und von ihr zurückgegeben werden. So baust du dir eigene Auswertungen:

k13_funktionen.py

```
1 def durchschnitt(werte):
2     return sum(werte) / len(werte)
3
4 def nur_gerade(zahlen):
5     ergebnis = []
```

```
    for zahl in zahlen:
        if zahl % 2 == 0:
            ergebnis.append(zahl)
    return ergebnis

puls = [72, 95, 120, 88, 131, 104]
print("Durchschnitt:", round(durchschnitt(puls), 1))
print("Gerade Werte:", nur_gerade(puls))
```

Terminal

```
Durchschnitt: 101.7
Gerade Werte: [72, 120, 88, 104]
```

- `durchschnitt` (Zeile 1–2) bekommt eine Liste als Parameter und liefert eine Zahl.
- `nur_gerade` (Zeile 4–9) durchläuft die Liste und baut dabei eine **neue** Liste, die sie zurückgibt. Die Liste `puls` bleibt unverändert.

Eine Funktion kann eine übergebene Liste aber auch **verändern**. Das liegt daran, dass eine Variable eine Liste nicht enthält, sondern nur auf sie **zeigt**. Zwei Namen können auf dieselbe Liste zeigen:

k13_gleiche_liste.py

```
def verdopple(werte):
    for i in range(len(werte)):
        werte[i] = werte[i] * 2

zahlen = [1, 2, 3]
verdopple(zahlen)
print("zahlen:", zahlen)

andere = zahlen   # keine Kopie!
andere.append(99)
print("zahlen:", zahlen)

kopie = zahlen[:]  # echte Kopie
kopie.append(5)
print("zahlen:", zahlen)
print("kopie: ", kopie)
```

Terminal

```
zahlen: [2, 4, 6]
zahlen: [2, 4, 6, 99]
zahlen: [2, 4, 6, 99]
kopie:  [2, 4, 6, 99, 5]
```

- Zeile 6: Der Parameter `werte` zeigt auf dieselbe Liste wie `zahlen`. Was `verdopple` mit `werte[i]` macht, sieht man danach auch in `zahlen`.
- Zeile 9: `andere = zahlen` kopiert die Liste **nicht**. `andere` ist nur ein zweiter Name für dieselbe Liste. Das `append` in Zeile 10 verändert deshalb auch `zahlen`.
- Zeile 13: Eine echte Kopie bekommst du mit `zahlen[:]` – ein Teilstück vom Anfang bis zum Ende. `kopie` ist eine eigene Liste; Änderungen daran betreffen `zahlen` nicht.

Ausprobieren: Die Bestenliste

Die Bestenliste zeigt die fünf höchsten Punktzahlen auf dem Display. Neue Punktzahlen tippst du im Terminal ein.



k13_bestenliste.py

```
1  import pbc
2
3  def zeige_bestenliste(liste, neu):
4      pbc.clear()
5      pbc.text24("Bestenliste", 52, 20, pbc.YELLOW)
6      for i in range(len(liste)):
7          farbe = pbc.WHITE
8          if liste[i] == neu:
9              farbe = pbc.GREEN
10         y = 70 + i * 35
11         pbc.text18(f"{i + 1}.", 50, y, farbe)
12         pbc.text18(str(liste[i]), 120, y, farbe)
13
14 punkte = [120, 95, 80]
15 zeige_bestenliste(punkte, -1)
16
17 while True:
18     eingabe = input("Neue Punktzahl (leer = Ende): ")
19     if eingabe == "":
20         break
21     neu = int(eingabe)
22     punkte.append(neu)
23     punkte = sorted(punkte, reverse=True)
24     if len(punkte) > 5:
25         punkte.pop()           # nur die besten 5 bleiben
26     if neu in punkte:
27         print(f"Platz {punkte.index(neu) + 1}!")
28     else:
29         print("Leider nicht in der Bestenliste.")
30     zeige_bestenliste(punkte, neu)
31
32 print("Endstand:", punkte)
```

- **Zeile 3–12:** `zeige_bestenliste` zeichnet die ganze Liste neu. Jeder Platz steht 35 Pixel unter dem vorigen. Ein Eintrag, der gleich `neu` ist, wird grün. Beim ersten Aufruf (Zeile 15) gibt es noch keine neue Punktzahl. Deshalb wird `-1` übergeben, ein Wert, der nie vorkommt.
- **Zeile 14:** Die Liste beginnt mit drei Einträgen.
- **Zeile 22–23:** Die neue Punktzahl wird angehängt und die Liste absteigend sortiert. Die beste Punktzahl steht danach bei Index 0.
- **Zeile 24–25:** Sind es mehr als fünf Einträge, fliegt mit `pop()` der letzte – also der schlechteste – heraus.
- **Zeile 26–29:** Ist die neue Punktzahl noch in der Liste, verrät `index` den Platz. Weil der Index bei 0 beginnt, kommt noch 1 dazu.

Zum Verändern: Zeige statt fünf nur drei Plätze. Gib die Plätze 1 bis 3 in Gold (`pbc.YELLOW`), Silber (`pbc.GRAY`) und Bronze (`pbc.ORANGE`) aus. Wie müsste das Programm aussehen, wenn **kleinere** Werte besser sind, zum Beispiel bei Laufzeiten?

Übungsaufgaben

Aufgabe 1 ⭐ Was gibt das Programm aus?
Schreibe die Ausgabe auf, ohne das Programm zu testen. Prüfe danach im Emulator.

k13_aufgabe1.py

```
1 farben = ["rot", "gelb", "blau"]
2 farben.append("gruen")
3 print(farben[1])
4 print(farben[-2])
5 print(len(farben))
6 weg = farben.pop(0)
7 print(weg, farben)
8 farben.insert(1, "lila")
9 print(farben)
```

Aufgabe 2 ⭐ Finde den Fehler Das Programm soll alle fünf Noten nummeriert ausgeben und dann den Schnitt. Es bricht aber ab, und eine Note fehlt. Was ist falsch? Korrigiere es.

k13_aufgabe2.py

```
1 noten = [2, 3, 1, 4, 2]
2
3 for i in range(1, len(noten) + 1):
4     print(f"Arbeit {i}: Note {noten[i]}")
5
6 print("Schnitt:", sum(noten) / len(noten))
```

Aufgabe 3 ⭐⭐ Würfelstatistik Würfle 30-mal mit `random.randint(1, 6)` und sammle die Würfe in einer Liste. Gib danach für jede Augenzahl von 1 bis 6 aus, wie oft sie gefallen ist, und den Durchschnitt aller Würfe. Tipp: `wuerfe.count(6)` zählt die Sechsen.

Aufgabe 4 ⭐⭐ Das Diagramm verbessern Ändere `k13_regen.py`: Der höchste Balken wird rot gezeichnet. Eine gelbe waagrechte Linie zeigt den Durchschnitt, und oben steht der Durchschnitt als Zahl.

Aufgabe 5 ⭐⭐ Eigene Auswertungen `spannweite(werte)` soll den Unterschied zwischen größtem und kleinstem Wert liefern, `ueber_durchschnitt(werte)` eine neue Liste mit allen Werten über dem Durchschnitt. Teste beide mit kleinen Listen, die du im Kopf ausrechnen kannst.

Aufgabe 6 ⭐⭐⭐ Notenspiegel Frage im Terminal so lange nach Noten (1 bis 6), bis eine leere Eingabe kommt. Zeige dann für jede Note einen Balken, dessen Länge der Anzahl entspricht, und im Terminal den Schnitt.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eine Liste speichert viele Werte unter einem Namen: `werte = [3, 1, 4]`. `liste[i]` holt das Element mit Index `i`. Der Index beginnt bei 0 und endet bei `len(liste) - 1`; `liste[-1]` ist das letzte Element.
- Verändern: `liste[i] = wert`, `append(wert)`, `insert(i, wert)`, `remove(wert)`, `pop()` bzw. `pop(i)`.
- Durchlaufen: `for wert in liste:` oder mit Index `for i in range(len(liste)):`.
- Auswerten: `len`, `sum`, `min`, `max`, `sorted` (auch `reverse=True` und `key=…`), `in`, `count`, `index`.
- Zwei Namen können auf dieselbe Liste zeigen. Eine Kopie gibt es mit `liste[:]`.

KAPITEL 14

Listen im Spiel

Worum geht's? In Spielen gibt es selten nur einen Gegner: Ufos, Sterne, Blasen und Schüsse kommen zu Dutzenden. Mit Listen bewegst du beliebig viele Objekte mit derselben Schleife. Du lernst, Spielfelder als Listen in Listen zu speichern, Objekte sicher aus einer Liste zu entfernen und Listen in einer einzigen Zeile zu erzeugen.

14.1 Viele Gegner: Positionen in Listen

Ein Ufo braucht drei Variablen: `x`, `y` und die Geschwindigkeit `dx`. Für fünf Ufos wären das 15 Variablen und fünfmal fast derselbe Programmcode. Besser: Du legst für jede Eigenschaft **eine Liste** an. Alles, was zum selben Ufo gehört, steht bei **demselben Index**. Solche Listen nennt man **parallele Listen**.

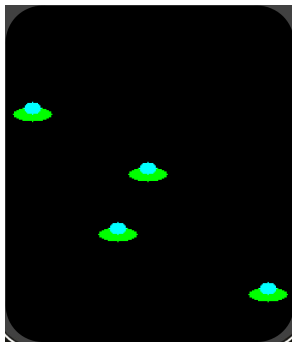
k14_ufos.py

```
import pbc
import time

# Index 0 gehoert zum ersten Ufo, Index 1 zum zweiten ...
xs = [30, 100, 170, 60, 140]
ys = [40, 90, 140, 190, 240]
dxs = [3, -2, 4, -3, 2]

def zeichne_ufo(x, y):
    pbc.ellipse(x, y, 16, 6, pbc.GREEN, fill=True)
    pbc.ellipse(x, y - 5, 7, 5, pbc.CYAN, fill=True)

pbc.clear()
while True:
    for i in range(len(xs)):
        pbc.fill_rect(xs[i] - 16, ys[i] - 10, 33, 17, pbc.BLACK)
        xs[i] += dxs[i]
        if xs[i] < 20 or xs[i] > 220:
            dxs[i] = -dxs[i]          # am Rand umkehren
        zeichne_ufo(xs[i], ys[i])
    time.sleep_ms(30)
```



- Zeile 5–7: Das erste Ufo hat `xs[0] = 30`, `ys[0] = 40` und `dxs[0] = 3`. Das zweite Ufo steht bei Index 1 und so weiter.
- Zeile 9–11: `zeichne_ufo` zeichnet ein Ufo aus zwei Ellipsen.
- Zeile 15: Die `for`-Schleife läuft in jedem Durchgang der Spielschleife über **alle** Ufos. `i` ist die Nummer des Ufos.
- Zeile 16: Zuerst wird das Ufo an der alten Stelle schwarz übermalt. Das Rechteck ist so groß, dass es das ganze Ufo bedeckt.
- Zeile 17–19: Das Ufo bewegt sich um `dxs[i]`. Am Rand wird die Richtung umgekehrt: Aus 3 wird −3 und umgekehrt.
- Zeile 20 zeichnet das Ufo an der neuen Stelle.

Ein sechstes Ufo? Hänge an jede der drei Listen einen Wert an. Am Rest des Programms änderst du nichts, denn `len(xs)` zählt mit.

Die Ufos flackern ein wenig, weil jedes kurz schwarz übermalt wird, bevor es neu erscheint (Kapitel 6.6). Flimmerfrei geht es mit dem Canvas aus Kapitel 17.

14.2 Listen in Listen: Spielfelder und Koordinaten

Parallele Listen haben einen Haken: Löschst du ein Ufo, musst du in **allen** Listen dasselbe Element entfernen. Vergisst du eine, passt nichts mehr zusammen. Die Lösung: Jedes Ufo wird **eine** kleine Liste `[x, y, dx]`, und alle Ufos stehen in einer großen Liste. Ein Element einer Liste kann nämlich selbst eine Liste sein.

```
# Jedes Ufo ist eine kleine Liste: [x, y, dx]
ufos = [[30, 40, 3], [100, 90, -2], [170, 140, 4],
        [60, 190, -3], [140, 240, 2]]
...
    for ufo in ufos:
        pbc.fill_rect(ufo[0] - 16, ufo[1] - 10, 33, 17, pbc.BLACK)
        ufo[0] += ufo[2]
        if ufo[0] < 20 or ufo[0] > 220:
            ufo[2] = -ufo[2]
        zeichne_ufo(ufo[0], ufo[1])
```

- `ufos[1]` ist die Liste `[100, 90, -2]`, also das zweite Ufo.
- `ufos[1][0]` ist davon das Element 0, die Zahl 100. Die erste Klammer wählt das Ufo, die zweite die Eigenschaft.
- Die Schleife `for ufo in ufos:` gibt dir jedes Ufo als kleine Liste. Weil `ufo` auf dieselbe Liste zeigt wie das Element in `ufos` (Kapitel 13.5), verändert `ufo[0] += ufo[2]` wirklich die Position in `ufos`.

Der Rest des Programms (`k14_ufos_listen.py`) ist derselbe wie in 14.1.

Tabellen: Auch zusammengehörende Daten wie Name und Punkte speicherst du so. Zum Sortieren sagst du `sorted` mit `key=`, **wonach** sortiert werden soll:

k14_rangliste.py

```
 1 def punkte_von(eintrag):
 2     return eintrag[1]
 3
 4 tabelle = [["Mia", 120], ["Ben", 95], ["Lea", 140]]
 5 tabelle.append(["Tom", 110])
 6
 7 print(tabelle[2])
 8 print(tabelle[2][1])
 9
10 rangliste = sorted(tabelle, key=punkte_von, reverse=True)
11 for i in range(len(rangliste)):
12     eintrag = rangliste[i]
13     print(f"{i + 1}. {eintrag[0]} {eintrag[1]}")
```

Terminal

```
['Lea', 140]
140
1. Lea 140
2. Mia 120
3. Tom 110
4. Ben 95
```

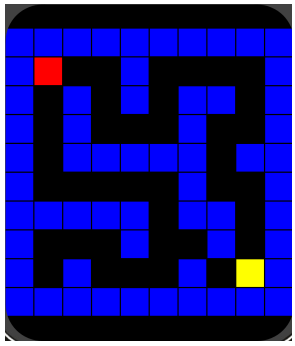
`punkte_von` (Zeile 1–2) liefert zu einem Eintrag die Punkte. `sorted` ruft diese Funktion für jeden Eintrag auf und sortiert nach dem Ergebnis (Zeile 10). Hinter `key=` steht der Funktionsname ohne Klammern.

Spielfelder: Ein Labyrinth besteht aus Zeilen, jede Zeile aus Feldern. Das ist eine Liste von Zeilen, jede Zeile eine Liste von Zahlen:

k14_labyrinth.py

```
 1 import pbc
 2
 3 # 0 = Weg, 1 = Mauer, 2 = Ziel
 4 karte = [
 5     [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 6     [1, 0, 0, 0, 1, 0, 0, 0, 0, 1],
 7     [1, 0, 1, 0, 1, 0, 1, 1, 0, 1],
 8     [1, 0, 1, 0, 0, 0, 1, 0, 0, 1],
 9     [1, 0, 1, 1, 1, 1, 1, 0, 1, 1],
10     [1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
11     [1, 1, 1, 1, 1, 0, 1, 1, 0, 1],
12     [1, 0, 0, 0, 1, 0, 0, 1, 0, 1],
13     [1, 0, 1, 0, 0, 0, 1, 0, 2, 1],
14     [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
15 ]
16 FARBEN = [pbc.BLACK, pbc.BLUE, pbc.YELLOW]
17 G = 24                    # ein Feld ist 24 x 24 Pixel
18
19 def zeichne_feld(zeile, spalte, farbe):
20     pbc.fill_rect(spalte * G, 20 + zeile * G, G - 1, G - 1, farbe)
21
22 pbc.clear()
```

```
23 for zeile in range(len(karte)):
24     for spalte in range(len(karte[zeile])):
25         zeichne_feld(zeile, spalte, FARBEN[karte[zeile][spalte]])
26
27 # Spielfigur links oben, Ziel rechts unten
28 zeichne_feld(1, 1, pbc.RED)
```



- Zeile 4–15: Die Karte hat 10 Zeilen mit je 10 Feldern. Untereinander geschrieben sieht man das Labyrinth schon im Programm. `karte[zeile][spalte]` ist ein Feld: erst die Zeile (von oben), dann die Spalte (von links). `karte[8][8]` ist das Ziel.
- Zeile 16: Auch die Farben stehen in einer Liste. `FARBEN[1]` ist die Farbe für eine Mauer. So wird aus der Zahl im Feld direkt eine Farbe (Zeile 25).
- Zeile 19–20: Ein Feld in Zeile `zeile` und Spalte `spalte` beginnt bei x = `spalte * 24` und y = `20 + zeile * 24`. Es wird 1 Pixel kleiner gezeichnet, damit man die Fugen sieht.
- Zeile 23–25: Zwei verschachtelte Schleifen zeichnen alle 100 Felder. `len(karte[zeile])` ist die Länge einer Zeile.

Soll eine Spielfigur durch das Labyrinth laufen, merkt sie sich ihre Zeile und Spalte. Vor jedem Schritt fragt das Programm `karte[zeile][spalte]` an der neuen Stelle ab: Ist dort eine 1, bleibt die Figur stehen (Aufgabe 4).

Ein Spielfeld nicht mit `*` vervielfachen. `[0] * 3` ist `[0, 0, 0]`. Dreimal dieselbe Zeile `[[0] * 3] * 3` enthält aber dreimal **dieselbe** Liste, nicht drei Kopien. Das Programm `k14_falle.py` setzt in beiden Varianten das Feld `[0][0]` auf 1 und gibt das Feld aus – oben mit `*`, unten mit drei einzeln angehängten Zeilen:

Terminal
```
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
[[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Ändert man ein Feld, ändert es sich in allen Zeilen. Lege jede Zeile einzeln an, mit `append` in einer Schleife oder wie in Abschnitt 14.4.

14.3 Elemente beim Durchlaufen entfernen

Sterne werden eingesammelt, Blasen platzen, Schüsse verlassen den Bildschirm: Im Spiel müssen ständig Elemente aus einer Liste verschwinden, und zwar mitten in der Schleife, die die Liste durchläuft. Genau dabei passiert ein tückischer Fehler:

k14_entfernen.py

```
 1 temperaturen = [12, -3, -5, 8, -1, 4]
 2
 3 # Falsch: Liste aendern, waehrend for laeuft
 4 werte = temperaturen[:]
 5 for t in werte:
 6     if t < 0:
 7         werte.remove(t)
 8 print("falsch:", werte)
 9
10 # Richtig 1: ueber eine Kopie laufen
11 werte = temperaturen[:]
12 for t in werte[:]:
13     if t < 0:
14         werte.remove(t)
15 print("Kopie:", werte)
16
17 # Richtig 2: rueckwaerts mit Index
18 werte = temperaturen[:]
19 for i in range(len(werte) - 1, -1, -1):
20     if werte[i] < 0:
21         werte.pop(i)
22 print("rueckwaerts:", werte)
```

Terminal

```
falsch: [12, -5, 8, 4]
Kopie: [12, 8, 4]
rueckwaerts: [12, 8, 4]
```

Was geht in Zeile 5–7 schief? Die `for`-Schleife merkt sich, beim wievielten Element sie ist. Wird `-3` (Index 1) entfernt, rücken alle folgenden Elemente eins nach vorn: `-5` steht jetzt bei Index 1. Die Schleife macht aber mit Index 2 weiter und **überspringt** `-5`. Es gibt keine Fehlermeldung, das Ergebnis ist einfach falsch.

Zwei sichere Wege:

1. **Über eine Kopie laufen** (Zeile 12): `werte[:]` ist eine Kopie. Die Schleife läuft über die Kopie, entfernt wird aus dem Original. Die Kopie bleibt unverändert, also wird nichts übersprungen.
2. **Rückwärts mit Index** (Zeile 19): `range(len(werte) - 1, -1, -1)` zählt von hinten nach vorn: 5, 4, 3, 2, 1, 0. Entfernt `pop(i)` ein Element, rücken nur die Elemente **dahinter** auf – und die hat die Schleife schon erledigt.

Im Spiel nimmst du meist den ersten Weg: `for stern in sterne[:]:`. Wie das aussieht, zeigt der Sternenregen am Ende des Kapitels. Ohne Entfernen würde die Liste dort immer länger, und das Programm würde mit der Zeit langsamer, weil es auch längst unsichtbare Sterne weiterbewegt.

14.4 Liste mit `range` erzeugen: List Comprehension

Oft soll eine Liste nach einer Regel gefüllt werden: 80 zufällige x-Werte, die Quadratzahlen, alle Werte über einer Grenze. Bisher brauchtest du dafür eine leere Liste, eine Schleife und `append`. Python hat dafür eine Kurzschreibweise, die **List Comprehension** (etwa: „Listen-Beschreibung“):

k14_comprehension.py

```
 1 import random
 2
 3 nullen = [0] * 5
 4 print(nullen)
 5
 6 zehner = [i * 10 for i in range(6)]
 7 print(zehner)
 8
 9 wuerfe = [random.randint(1, 6) for i in range(10)]
10 print(wuerfe)
11
12 grosse = [w for w in wuerfe if w >= 5]
13 print(grosse)
14
15 celsius = [10, 20, 25]
16 fahrenheit = [c * 9 / 5 + 32 for c in celsius]
17 print(fahrenheit)
18
19 gitter = [[0] * 3 for i in range(3)]
20 gitter[0][0] = 1
21 print(gitter)
```

Terminal

```
[0, 0, 0, 0, 0]
[0, 10, 20, 30, 40, 50]
[3, 5, 5, 3, 3, 2, 4, 1, 3, 5]
[5, 5, 5]
[50.0, 68.0, 77.0]
[[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

- Zeile 3: `[0] * 5` wiederholt den Inhalt fünfmal. Das ist die einfachste Art, eine Liste mit lauter gleichen Zahlen zu bekommen.
- Zeile 6: `[i * 10 for i in range(6)]` liest du so: „die Liste aller `i * 10`, wobei `i` von 0 bis 5 läuft“. Vorn steht, **was** in die Liste kommt, hinten die `for`-Schleife.
- Zeile 9: Vorn darf jeder Ausdruck stehen, auch ein Zufallswert. Die Variable `i` wird hier gar nicht benutzt, sie zählt nur zehn Durchgänge.
- Zeile 12: Mit `if` am Ende kommen nur die Elemente in die neue Liste, für die die Bedingung gilt.
- Zeile 15–16: Die Schleife kann auch über eine andere Liste laufen. Aus jeder Celsius-Temperatur wird eine Fahrenheit-Temperatur.
- Zeile 19: So legst du ein Spielfeld richtig an: Für jede Zeile wird `[0] * 3` neu ausgewertet, jede Zeile ist eine eigene Liste.

Zeile 12 ist zugleich die einfachste Art, Elemente zu entfernen: Statt aus der alten Liste zu löschen, baust du eine neue Liste nur mit den Elementen, die bleiben sollen.

Ausprobieren: Der Sternenregen

Sterne fallen vom Himmel. Fang sie mit der Schale auf, die du mit dem Joystick nach links und rechts bewegst. Jeder gefangene Stern bringt einen Punkt, jeder verpasste kostet ein Leben.

k14_sternenregen.py

```
import pbc
import time
import random

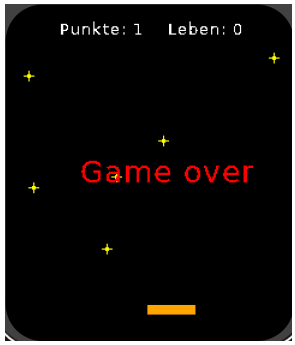
SCHALE_Y = 250            # Oberkante der Schale
BREITE = 40               # Breite der Schale
schale_x = 100
sterne = []               # jeder Stern: [x, y, tempo]
punkte = 0
leben = 3

def zeichne_stern(x, y, farbe):
    pbc.hline(x - 4, y, 9, farbe)
    pbc.vline(x, y - 4, 9, farbe)
    pbc.fill_rect(x - 1, y - 1, 3, 3, farbe)

def zeige_stand():
    pbc.fill_rect(0, 0, 240, 30, pbc.BLACK)
    pbc.text12(f"Punkte: {punkte}    Leben: {leben}", 45, 12, pbc.WHITE)

pbc.clear()
zeige_stand()
while leben > 0:
    # Schale bewegen
    pbc.fill_rect(schale_x, SCHALE_Y, BREITE, 8, pbc.BLACK)
    if pbc.pressed_left() and schale_x > 5:
        schale_x -= 6
    if pbc.pressed_right() and schale_x < 235 - BREITE:
        schale_x += 6
    pbc.fill_rect(schale_x, SCHALE_Y, BREITE, 8, pbc.ORANGE)

    # Ab und zu ein neuer Stern
    if random.randint(1, 12) == 1:
        sterne.append([random.randint(10, 230), 35, random.randint(2, 5)])
```



- **Zeile 5–10:** Konstanten für die Schale, die Liste `sterne` (anfangs leer), Punkte und Leben.
- **Zeile 12–19:** Ein Stern ist ein kleines Kreuz aus zwei Linien. Mit `pbc.BLACK` wird er gelöscht. `zeige_stand` schreibt die Statuszeile neu.
- **Zeile 25–30:** Die Schale wird übermalt, bewegt und neu gezeichnet. `pressed_left()` gilt, solange die Taste gedrückt ist. Die Bedingungen hinter `and` halten die Schale am Rand.
- **Zeile 33–34:** Ungefähr in jedem zwölften Durchgang kommt ein neuer Stern mit zufälliger Position und Geschwindigkeit dazu.

k14_sternenregen.py

```
35
36      for stern in sterne[:]:
37          zeichne_stern(stern[0], stern[1], pbc.BLACK)
38          stern[1] += stern[2]
39          if stern[1] >= SCHALE_Y - 4:      # unten angekommen
40              sterne.remove(stern)
41              if schale_x - 4 <= stern[0] and stern[0] <= schale_x + BREITE + 4:
42                  punkte += 1
43              else:
44                  leben -= 1
45              zeige_stand()
46          else:
47              zeichne_stern(stern[0], stern[1], pbc.YELLOW)
48      time.sleep_ms(30)
49
50  pbc.text24("Game over", 62, 125, pbc.RED)
```

- **Zeile 36:** Die Schleife läuft über eine Kopie, denn in Zeile 40 werden Sterne entfernt.
- **Zeile 39–45:** Ist der Stern unten angekommen, wird er auf jeden Fall entfernt. Liegt sein x im Bereich der Schale (mit 4 Pixeln Spielraum), gibt es einen Punkt, sonst geht ein Leben verloren.
- **Zeile 23 und 50:** Die Spielschleife läuft, solange noch Leben übrig sind. Danach erscheint „Game over“.

Zum Verändern: Mach die Schale breiter oder schmaler. Lass mehr Sterne regnen (`randint(1, 6)`). Gib jedem Stern eine zufällige Farbe als viertes Element `stern[3]` und zeichne ihn damit.

Übungsaufgaben

Aufgabe 1 ⭐ Was gibt das Programm aus? Schreibe die Ausgabe auf, ohne das Programm zu testen. Prüfe danach im Emulator.

k14_aufgabe1.py

```
1  feld = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
2  print(feld[1])
3  print(feld[2][0])
4  print(feld[0][-1])
5  print(len(feld), len(feld[0]))
6
7  summe = 0
8  for zeile in feld:
9      summe += zeile[1]
10 print(summe)
11
12 print([zeile[0] for zeile in feld])
```

Aufgabe 2 ⭐ Finde den Fehler Das Programm soll alle Nullen aus der Liste entfernen. Es läuft ohne Fehlermeldung, aber eine Null bleibt übrig. Erkläre, warum, und korrigiere es auf zwei verschiedene Arten.

k14_aufgabe2.py

```
1 punkte = [5, 0, 0, 7, 0, 3]
2 for p in punkte:
3     if p == 0:
4         punkte.remove(p)
5 print(punkte)
```

Aufgabe 3 ⭐⭐ Schneefall Lass 40 Schneeflocken (3 × 3 Pixel, weiß) fallen. Erzeuge x, y und die Geschwindigkeit (1 bis 4) jeder Flocke mit List Comprehensions. Verlässt eine Flocke unten den Bildschirm, beginnt sie oben an einer zufälligen x-Position neu.

Aufgabe 4 ⭐⭐ Durchs Labyrinth Erweitere `k14_labyrinth.py`: Die rote Figur läuft mit dem Joystick (`was_pressed_…`) Feld für Feld, nicht durch Mauern. Eine 3 in der Karte ist eine Münze (orange). Beim Einsammeln wird das Feld zu 0. Am Ziel steht, wie viele Münzen von wie vielen gesammelt wurden (Tipp: `count(3)`).

Aufgabe 5 ⭐⭐ Listen in einer Zeile Erzeuge mit je einer List Comprehension: die Quadratzahlen bis 100 sowie aus `namen = ["Ida", "Konstantin", "Ben", "Marie", "Lu"]` die Längen, die Namen mit mehr als drei Buchstaben und die Namen in Großbuchstaben.

Aufgabe 6 ⭐⭐⭐ Meteore im Sternenregen Erweitere den Sternenregen: Jeder vierte fallende Gegenstand ist ein roter Meteor. Wer einen Meteor fängt, verliert ein Leben; ein verpasster Meteor schadet nicht. Je mehr Punkte, desto schneller fallen neue Sterne.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Viele Objekte: parallele Listen (gleicher Index = gleiches Objekt) oder eine Liste kleiner Listen `[x, y, …]`, bewegt mit einer `for`-Schleife.
- Spielfelder sind Listen von Zeilen: `karte[zeile][spalte]`.
- Nie aus der Liste entfernen, über die die Schleife gerade läuft. Über eine Kopie laufen (`liste[:]`), rückwärts mit Index oder eine neue Liste bauen.
- List Comprehension: `[ausdruck for i in range(n)]`, mit Filter `[x for x in liste if bedingung]`.
- `sorted(liste, key=funktion)` sortiert nach dem, was die Funktion liefert.

KAPITEL 15

Dateien: Daten dauerhaft speichern

Worum geht's? Bisher vergisst dein Programm alles, sobald es endet: jeden Punktestand, jede Eingabe, jede Liste. In diesem Kapitel speicherst du Daten in einer **Datei**. Beim nächsten Start liest das Programm sie wieder ein. So bleibt ein Highscore erhalten, auch wenn du den PicoBoy ausschaltest.

15.1 Eine Datei schreiben und lesen

Eine **Datei** ist ein Stück Speicher mit einem Namen, zum Beispiel `gruss.txt`. Deine Programme, die du mit **Dateien → Auf den PBC+...** auf das Gerät legst, sind auch Dateien. Ein Programm kann selbst Dateien anlegen, hineinschreiben und daraus lesen. Die Endung `.txt` zeigt: In der Datei steht einfacher Text.

Die Arbeit mit einer Datei hat immer drei Schritte: **öffnen**, **schreiben oder lesen**, **schließen**.

k15_schreiben.py

```
1 # Schreibt einen Gruß in die Datei gruss.txt
2 datei = open("gruss.txt", "w")
3 datei.write("Viele Grüße vom PicoBoy!")
4 datei.close()
5 print("Gespeichert.")
```

Terminal

```
Gespeichert.
```

- Zeile 2: `open("gruss.txt", "w")` öffnet die Datei `gruss.txt`. Das zweite Argument ist der **Modus**: `"w"` (englisch *write*) heißt „zum Schreiben". Gibt es die Datei noch nicht, wird sie angelegt. Gibt es sie schon, wird ihr alter Inhalt **gelöscht**. `open` liefert ein **Dateiobjekt**; wir speichern es in der Variablen `datei`.
- Zeile 3: `datei.write(…)` schreibt einen Text in die Datei.
- Zeile 4: `datei.close()` schließt die Datei. Erst jetzt ist sicher, dass alles gespeichert ist.

Auf dem Bildschirm und in der Konsole passiert dabei fast nichts. Ob es geklappt hat, zeigt ein **zweites** Programm, das die Datei liest:

k15_lesen.py

```
# Liest die Datei, die k15_schreiben.py angelegt hat
datei = open("gruss.txt", "r")
inhalt = datei.read()
datei.close()
print("In der Datei steht:")
print(inhalt)
```

Terminal

```
In der Datei steht:
Viele Grüße vom PicoBoy!
```

Der Modus `"r"` (*read*) öffnet die Datei zum Lesen. `datei.read()` liefert den ganzen Inhalt als String. Das Programm `k15_lesen.py` weiß nichts von der Variablen aus `k15_schreiben.py`. Der Text kommt wirklich aus der Datei.

Die Datei gibt es nicht. Startest du ein Programm, das eine Datei lesen will, die es nicht gibt, bricht es mit einem `OSError` ab:

k15_fehler_fehlt.py

```
datei = open("notizen.txt", "r")
inhalt = datei.read()
datei.close()
print(inhalt)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] ENOENT
```

`ENOENT` steht für *error, no entry*: Es gibt keinen Eintrag mit diesem Namen. PBC+ Studio zeigt dazu das Hinweisfenster „Datei nicht gefunden“. Prüfe den Dateinamen. Wie ein Programm vorher nachsieht, ob es die Datei gibt, lernst du in Abschnitt 15.4.

Zahlen speichern. In eine Textdatei kann man nur Strings schreiben. Eine Zahl wandelst du deshalb vorher mit `str()` in Text um. Beim Lesen bekommst du wieder Text und machst mit `int()` eine Zahl daraus:

k15_rekord.py

```
rekord = 412    # Weitsprung in cm

# Zahl als Text speichern
datei = open("rekord.txt", "w")
datei.write(str(rekord))
```

```
 6 datei.close()
 7
 8 # Text lesen und wieder in eine Zahl umwandeln
 9 datei = open("rekord.txt", "r")
10 gelesen = int(datei.read())
11 datei.close()
12
13 print("Rekord:", gelesen, "cm")
14 print("Mit 10 cm mehr:", gelesen + 10, "cm")
```

Terminal

```
Rekord: 412 cm
Mit 10 cm mehr: 422 cm
```

In Zeile 5 wird aus der Zahl `412` der Text `"412"`. In Zeile 10 passiert das Umgekehrte: `datei.read()` liefert `"412"`, `int()` macht daraus die Zahl `412`. Nur mit der Zahl kann Zeile 14 rechnen.

Vergisst du `str()` und schreibst in Zeile 5 nur `datei.write(rekord)` (`k15_fehler_zahl.py`), meldet `write` einen `TypeError`:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: object with buffer protocol required
```

Die Meldung ist schwer zu verstehen. Merke dir einfach: `TypeError` in einer Zeile mit `write` bedeutet fast immer, dass dort eine Zahl statt eines Strings steht.

15.2 Zeile für Zeile lesen

Oft will man mehrere Werte speichern, etwa eine Liste. Dann schreibt man jeden Wert in eine eigene **Zeile**. Das Zeichen für „neue Zeile“ ist `\n` (ein Rückwärtsschrägstrich und ein n). Es zählt als **ein** Zeichen und steht am Ende jeder Zeile:

k15_gedicht.py

```
 1 # Drei Zeilen schreiben: jede endet mit \n
 2 datei = open("gedicht.txt", "w")
 3 datei.write("Der Mond ist aufgegangen,\n")
 4 datei.write("die goldnen Sternlein prangen\n")
 5 datei.write("am Himmel hell und klar.\n")
 6 datei.close()
 7
 8 datei = open("gedicht.txt", "r")
 9 erste = datei.readline()
10 print("Die erste Zeile:", erste.strip())
11 nummer = 2
```

```
12 for zeile in datei:
13     print(nummer, zeile.strip())
14     nummer += 1
15 datei.close()
```

Terminal

```
Die erste Zeile: Der Mond ist aufgegangen,
2 die goldnen Sternlein prangen
3 am Himmel hell und klar.
```

- Zeile 2–6: Jeder `write`-Befehl schreibt eine Zeile, die mit `\n` endet. Ohne `\n` stünde alles in einer einzigen langen Zeile.
- Zeile 9: `datei.readline()` liest **eine** Zeile, und zwar die nächste, die noch nicht gelesen wurde.
- Zeile 12: `for zeile in datei:` läuft Zeile für Zeile durch den **Rest** der Datei, genau wie eine `for`-Schleife durch eine Liste. Die erste Zeile kommt nicht noch einmal: Sie wurde schon gelesen.
- `zeile.strip()` entfernt Leerzeichen und das `\n` am Anfang und Ende eines Strings.

Warum `strip()`? Jede gelesene Zeile endet mit `\n`, und `print` hängt noch ein `\n` an. Lässt du in Zeile 10 und 13 `.strip()` weg (`k15_ohne_strip.py`), steht nach jeder Zeile eine leere Zeile:

Terminal

```
Die erste Zeile: Der Mond ist aufgegangen,

2 die goldnen Sternlein prangen

3 am Himmel hell und klar.
```

Eine Liste speichern und wieder einlesen. Das folgende Programm speichert die Temperaturen einer Woche, eine Zahl pro Zeile. Danach liest es die Datei und baut daraus wieder eine Liste:

k15_wetter.py

```
 1 temperaturen = [12, 15, 14, 9, 11, 16, 18]
 2
 3 # Jede Zahl in eine eigene Zeile
 4 datei = open("wetter.txt", "w")
 5 for wert in temperaturen:
 6     datei.write(str(wert) + "\n")
 7 datei.close()
 8
 9 # Zeilen lesen und wieder eine Liste daraus bauen
10 gelesen = []
11 datei = open("wetter.txt", "r")
12 for zeile in datei:
13     gelesen.append(int(zeile))
```

```
14 datei.close()
15
16 print("Gelesen:", gelesen)
17 print("Wärmster Tag:", max(gelesen), "Grad")
18 print("Mittelwert:", round(sum(gelesen) / len(gelesen), 1), "Grad")
```

Terminal

```
Gelesen: [12, 15, 14, 9, 11, 16, 18]
Wärmster Tag: 18 Grad
Mittelwert: 13.6 Grad
```

- Zeile 5–6: Die Schleife schreibt jede Zahl als Text mit `\n` dahinter. In der Datei steht also `12`, `15`, `14` … untereinander.
- Zeile 10–13: Wir beginnen mit einer leeren Liste und hängen für jede gelesene Zeile die Zahl an. `int("12\n")` ergibt `12`: `int()` übersieht das `\n` am Ende, `strip()` ist hier nicht nötig.
- Zeile 16–18: Mit der gelesenen Liste rechnest du wie mit jeder anderen Liste.

15.3 Mit `with` arbeiten

Das `close()` am Ende vergisst man leicht. Schlimmer: Bricht das Programm vorher mit einem Fehler ab, bleibt die Datei offen, und der zuletzt geschriebene Text fehlt unter Umständen. Python hat dafür eine bequeme Schreibweise:

```
with open("wetter.txt", "r") as datei:
    inhalt = datei.read()
print(inhalt)
```

`with open(…) as datei:` öffnet die Datei und speichert das Dateiobjekt in `datei`. Alles, was **eingerückt** darunter steht, arbeitet mit der Datei. Sobald der eingerückte Block zu Ende ist, schließt Python die Datei von selbst, auch wenn im Block ein Fehler passiert. Ab jetzt benutzen wir immer `with`.

Anhängen statt überschreiben. Der Modus `"w"` löscht den alten Inhalt. Für ein Tagebuch, in das jeden Tag etwas dazukommt, gibt es den Modus `"a"` (*append*, anhängen): Neuer Text kommt ans Ende der Datei, das Alte bleibt stehen.

k15_training.py

```
1 sportart = input("Was hast du heute trainiert? ")
2 minuten = input("Wie viele Minuten? ")
3
4 # "a": neue Zeile hinten anhängen, Altes bleibt stehen
5 with open("training.txt", "a") as datei:
6     datei.write(sportart + ": " + minuten + " min\n")
```

```
 8 print("Dein Trainingstagebuch:")
 9 with open("training.txt", "r") as datei:
10     for zeile in datei:
11         print("-", zeile.strip())
```

Terminal

```
Was hast du heute trainiert? Laufen
Wie viele Minuten? 30
Dein Trainingstagebuch:
- Laufen: 30 min
```

- Zeile 5–6: Die Datei wird im Modus `"a"` geöffnet. Gibt es sie noch nicht, wird sie angelegt. Die neue Zeile kommt ans Ende.
- Zeile 9–11: Danach liest das Programm die ganze Datei und zeigt alle Einträge. Startest du es noch einmal, steht der erste Eintrag immer noch da, und der neue kommt darunter.

Modus	Bedeutung	Gibt es die Datei nicht …
`"r"`	lesen (auch, wenn du den Modus weglässt)	`OSError`
`"w"`	schreiben, alter Inhalt wird gelöscht	wird sie angelegt
`"a"`	anhängen, alter Inhalt bleibt	wird sie angelegt

15.4 Dateien auf dem Gerät und im Emulator

Auf dem Gerät landen die Dateien im eingebauten Speicher des PicoBoy, an derselben Stelle wie deine Programme. Sie bleiben dort, auch wenn du den PicoBoy ausschaltest oder ein anderes Programm startest. Unter **Dateien → Dateien verwalten…** siehst du alle Dateien auf dem Gerät und kannst einzelne löschen.

Im Emulator bleiben Dateien nur erhalten, bis du die Seite neu lädst oder schließt. Danach sind alle selbst angelegten Dateien weg. Zum Ausprobieren reicht das. Willst du aber testen, ob dein Highscore wirklich „überlebt“, brauchst du das Gerät.

> Der Speicher des PicoBoy verträgt sehr viele, aber nicht unendlich viele Schreibvorgänge. Schreibe deshalb nicht in jedem Durchlauf einer Spielschleife in eine Datei, sondern nur, wenn es nötig ist, zum Beispiel einmal am Spielende.

Welche Dateien gibt es? Das Modul `os` (*operating system*, Betriebssystem) kennt Befehle für Dateien. `os.listdir()` liefert eine Liste mit den Namen aller Dateien:

k15_dateiliste.py

```
1 import os
2
3 print(os.listdir())
```

Terminal (Emulator)

```
['tmp', 'home', 'dev', 'proc', 'lib', 'farben.txt', 'muenzen.txt']
```

Hinten stehen die Dateien, die die Programme dieses Kapitels angelegt haben. Die ersten fünf Namen (`tmp`, `home`, `dev`, `proc`, `lib`) sind Ordner, die nur der Emulator hat. Auf dem Gerät stehen in der Liste stattdessen auch die Programme, die du dort gespeichert hast.

Vorher nachsehen. Weil `os.listdir()` eine Liste liefert, kannst du mit `in` prüfen, ob eine Datei da ist. So vermeidet ein Programm den `OSError` aus Abschnitt 15.1. Dieses Programm zählt, wie oft es schon gestartet wurde:

k15_starts.py

```
 1 import os
 2 import pbc
 3
 4 DATEI = "starts.txt"
 5
 6 # Gibt es die Datei schon? Sonst ist dies der erste Start.
 7 if DATEI in os.listdir():
 8     with open(DATEI, "r") as datei:
 9         starts = int(datei.read())
10 else:
11     starts = 0
12
13 starts += 1
14 with open(DATEI, "w") as datei:
15     datei.write(str(starts))
16
17 print("Start Nummer", starts)
18 pbc.clear()
19 pbc.text18("Programmstarts:", 35, 100, pbc.WHITE)
20 pbc.text24(str(starts), 112, 140, pbc.YELLOW)
```



- Zeile 7: Steht `starts.txt` in der Liste, liest das Programm die bisherige Zahl (Zeile 8–9). Sonst ist es der erste Start, und `starts` beginnt bei 0 (Zeile 11).
- Zeile 13–15: Eins dazuzählen und die neue Zahl speichern. Der Modus `"w"` überschreibt die alte Zahl.
- Das Bild zeigt den zweiten Start. Lädst du die Seite neu, beginnt der Emulator wieder bei 1; auf dem Gerät zählt er immer weiter.

Dieses Muster (nachsehen – lesen oder Startwert – ändern – speichern) brauchst du für jeden Highscore.

Eine Datei löschen. `os.remove("name")` löscht eine Datei. Auch hier prüft man vorher, ob es sie gibt, sonst kommt wieder ein `OSError`:

k15_loeschen.py

```
import os

DATEI = "starts.txt"

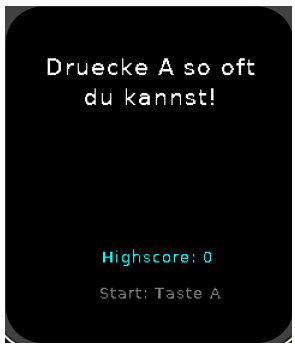
if DATEI in os.listdir():
    os.remove(DATEI)
    print(DATEI, "gelöscht.")
else:
    print(DATEI, "gibt es nicht.")
```

Terminal

```
starts.txt gibt es nicht.
```

Ausprobieren: Der Highscore bleibt

Wie oft schaffst du es, in fünf Sekunden Taste A zu drücken? Das Programm merkt sich den besten Wert in der Datei `highscore.txt` und zeigt ihn auf dem Startbildschirm.



k15_highscore.py

```
import os
import pbc
import time

DATEI = "highscore.txt"
SPIELZEIT = 5000    # Millisekunden

def lade_highscore():
    if DATEI in os.listdir():
        with open(DATEI, "r") as datei:
            return int(datei.read())
    return 0        # noch kein Highscore gespeichert

def speichere_highscore(wert):
    with open(DATEI, "w") as datei:
        datei.write(str(wert))

def zeige_zahl(zahl):
    pbc.fill_rect(80, 120, 80, 40, pbc.BLACK)
    pbc.text24(str(zahl), 105, 125, pbc.YELLOW)
```

Die Arbeit mit der Datei steckt in zwei Funktionen:

- `lade_highscore()` (Zeile 8–12) sieht nach, ob es die Datei gibt, und liefert den gespeicherten Wert. Gibt es noch keine Datei, liefert sie `0`. Das `return` in Zeile 11 steht im `with`-Block. Das ist erlaubt: Python schließt die Datei trotzdem.
- `speichere_highscore(wert)` (Zeile 14–16) schreibt einen neuen Wert und überschreibt den alten.
- `zeige_zahl` (Zeile 18–20) übermalt die alte Zahl mit einem schwarzen Rechteck und schreibt die neue.

k15_highscore.py

```
22 highscore = lade_highscore()
23 pbc.clear()
24 pbc.text18("Druecke A so oft", 33, 40, pbc.WHITE)
25 pbc.text18("du kannst!", 65, 65, pbc.WHITE)
26 pbc.text12(f"Highscore: {highscore}", 80, 200, pbc.CYAN)
27 pbc.text12("Start: Taste A", 78, 230, pbc.GRAY)
28 while not pbc.was_pressed_a():
29     time.sleep_ms(20)
30
31 treffer = 0
32 zeige_zahl(treffer)
33 start = time.ticks_ms()
34 while time.ticks_diff(time.ticks_ms(), start) < SPIELZEIT:
35     if pbc.was_pressed_a():
36         treffer += 1
37         zeige_zahl(treffer)
38     time.sleep_ms(10)
39
40 pbc.clear()
41 pbc.text24(f"{treffer} mal!", 80, 90, pbc.YELLOW)
42 if treffer > highscore:
43     speichere_highscore(treffer)
44     pbc.text18("Neuer Highscore!", 30, 150, pbc.GREEN)
45 else:
46     pbc.text18(f"Highscore: {highscore}", 60, 150, pbc.WHITE)
47 print("Treffer:", treffer, " Highscore vorher:", highscore)
```



- Zeile 22: Der Highscore wird **einmal** am Anfang geladen und auf dem Startbildschirm gezeigt.
- Zeile 31–38: Die Spielschleife zählt fünf Sekunden lang jeden Druck auf A (Kapitel 8).
- Zeile 42–44: Nur wenn der alte Highscore übertroffen wurde, wird die Datei neu geschrieben, also höchstens einmal pro Spiel.

Teste das Spiel mehrmals hintereinander: Der Highscore auf dem Startbildschirm wächst mit. Auf dem Gerät bleibt er auch nach dem Ausschalten erhalten.

Zum Verändern: Ändere die Spielzeit auf drei Sekunden. Lösche den Highscore mit einem kleinen Programm wie `k15_loeschen.py`. Speichere zusätzlich, wie oft insgesamt gespielt wurde.

Übungsaufgaben

Aufgabe 1 ⭐ Was steht in der Datei? Lies das Programm, ohne es zu testen. Was gibt es aus? Prüfe danach im Emulator.

k15_aufgabe1.py

```
with open("farben.txt", "w") as datei:
    datei.write("rot\n")
    datei.write("gruen\n")

with open("farben.txt", "w") as datei:
    datei.write("blau\n")

with open("farben.txt", "a") as datei:
    datei.write("gelb\n")

with open("farben.txt", "r") as datei:
    for zeile in datei:
        print(zeile.strip())
```

Aufgabe 2 ⭐ Finde die Fehler Das Programm soll die Zahl der gesammelten Münzen speichern, wieder lesen und 3 Bonusmünzen dazuzählen. Es enthält zwei Fehler. Teste, lies die Fehlermeldung, korrigiere und teste erneut.

k15_aufgabe2.py

```
muenzen = 7

with open("muenzen.txt", "w") as datei:
    datei.write(muenzen)

with open("muenzen.txt", "r") as datei:
    gespeichert = datei.read()

print("Mit Bonus:", gespeichert + 3)
```

Aufgabe 3 ⭐⭐ Lieblingsfilme Schreibe ein Programm, das mit `input` nach drei Lieblingsfilmen fragt und sie in die Datei `filme.txt` schreibt, einen pro Zeile. Danach liest es die Datei und gibt die Filme nummeriert aus: `1. …`, `2. …`, `3. …`.

Aufgabe 4 ⭐⭐ Willkommen zurück Beim ersten Start fragt das Programm nach deinem Namen, speichert ihn in `name.txt` und zeigt auf dem Display „Hallo, …!“. Bei jedem weiteren Start fragt es nicht mehr, sondern zeigt „Willkommen zurueck, …!“.

Aufgabe 5 ⭐⭐⭐ Würfelstatistik über viele Starts Ein Programm würfelt bei jedem Start 60-mal und zählt, wie oft jede Augenzahl kommt. Die Zählung soll über alle Starts hinweg wachsen: Speichere die sechs Zahlen in einer Datei (eine pro Zeile), lies sie beim nächsten Start ein und zähle weiter. Gib am Ende die Gesamtzahl der Würfe und die sechs Anzahlen aus.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `open(name, modus)` öffnet eine Datei: `"r"` lesen, `"w"` schreiben (alter Inhalt wird gelöscht), `"a"` anhängen.
- Schreiben mit `datei.write(text)`, lesen mit `datei.read()` (alles), `datei.readline()` (eine Zeile) oder `for zeile in datei:`.
- In Dateien steht Text: Zahlen beim Schreiben mit `str()`, beim Lesen mit `int()` umwandeln. Jede Zeile endet mit `\n`; `strip()` entfernt es.
- `with open(…) as datei:` schließt die Datei am Ende des Blocks von selbst.
- `import os`: `os.listdir()` liefert die Liste der Dateien, `"name" in os.listdir()` prüft, ob es eine Datei gibt, `os.remove(name)` löscht sie.
- Auf dem Gerät bleiben Dateien dauerhaft, im Emulator nur bis zum Neuladen der Seite.

KAPITEL 16

Turtle-Grafik

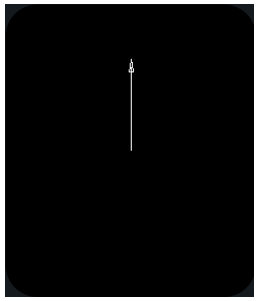
Worum geht's? Eine kleine Schildkröte mit einem Stift läuft über den Bildschirm und zeichnet dabei. Du steuerst sie mit Befehlen wie „geh 50 Pixel vorwärts“ und „dreh dich um 90 Grad“. Mit Schleifen und Funktionen entstehen so Vielecke, Muster und sogar ein Baum. Außerdem lernst du, wie die Turtle Farben vor sich erkennt.

16.1 Eine Schildkröte mit Stift: `turtle.Turtle()`

„Turtle“ ist Englisch und heißt Schildkröte. Die Turtle-Befehle stecken im Modul `turtle`, das du wie `pbc` mit `import` holst:

k16_erste.py

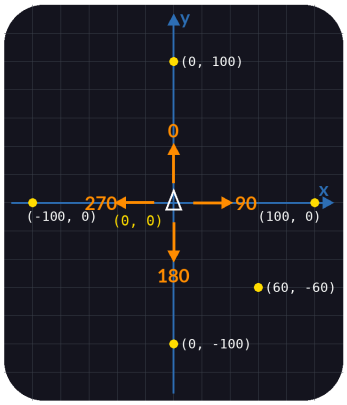
```
1 import turtle
2
3 t = turtle.Turtle()
4 t.forward(80)
```



- Zeile 3 legt mit `turtle.Turtle()` eine neue Turtle an und speichert sie in der Variablen `t`. Dabei wird der Bildschirm schwarz, und in der Mitte erscheint ein kleines Dreieck: die Turtle. Die Spitze zeigt, wohin sie schaut. Am Anfang schaut sie nach oben.
- Zeile 4 schickt die Turtle 80 Pixel vorwärts. Ihr Stift ist unten, sie zeichnet also eine weiße Linie.

Die Turtle ist ein **Objekt**: ein Ding, das Werte (Position, Richtung, Farbe) und eigene Befehle hat. Die Befehle eines Objekts heißen **Methoden**. Du rufst sie mit einem Punkt hinter dem Namen des Objekts auf: `t.forward(80)`. Das kennst du schon von Listen, zum Beispiel `punkte.append(5)`.

> **Die Turtle rechnet anders als `pbc`.** Für die Turtle liegt der Punkt (0, 0) in der **Mitte** des Bildschirms, und y wächst nach **oben**, wie im Matheunterricht. x geht von −120 (links) bis 119 (rechts), y von −139 (unten) bis 140 (oben).



16.2 Vorwärts, rückwärts, drehen

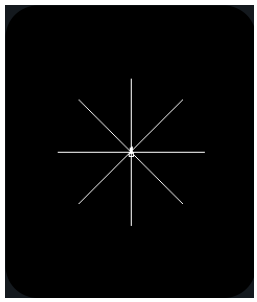
Vier Methoden bewegen die Turtle: `t.forward(d)` geht `d` Pixel vorwärts in Blickrichtung, `t.back(d)` rückwärts (die Blickrichtung bleibt). `t.right(w)` dreht sie um `w` Grad nach rechts (im Uhrzeigersinn), `t.left(w)` nach links. Beim Drehen bleibt die Turtle an ihrem Platz. Eine

Vierteldrehung sind 90 Grad, eine halbe 180 Grad, eine ganze 360 Grad.

`right` und `left` drehen die Turtle **um** einen Winkel, ausgehend von ihrer jetzigen Richtung. Mit `t.setDirection(w)` stellst du dagegen die Richtung direkt ein, egal wohin sie vorher geschaut hat: `0` heißt oben, `90` rechts, `180` unten und `270` links (siehe Abbildung in 16.1). So zeichnet die Turtle eine Windrose:

k16_windrose.py

```
1 import turtle
2
3 t = turtle.Turtle()
4 for richtung in [0, 90, 180, 270, 45, 135, 225, 315]:
5     t.setDirection(richtung)
6     t.forward(70)
7     t.back(70)
8 t.setDirection(0)
```



Zeile 5 richtet die Turtle nacheinander in die acht Richtungen aus. `back` bringt sie danach rückwärts zurück in die Mitte, ohne dass sie sich umdrehen muss. 45 ist die Richtung genau zwischen oben und rechts.

Vergisst du beim Drehen den Winkel, meldet Python einen `TypeError`:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError: left() takes 2 positional arguments but 1 were given
```

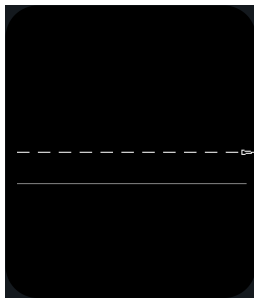
Die Meldung sagt „2 Argumente erwartet, aber 1 bekommen“, obwohl du gar keins angegeben hast. Python zählt hier die Turtle selbst mit. Gemeint ist: In Zeile 5 fehlt der Winkel, also z. B. `t.left(90)`.

16.3 Stift, Farbe, Geschwindigkeit

Methode	Bedeutung
`t.penUp()`, `t.penDown()`	Stift heben (nicht zeichnen) / wieder senken
`t.setPosition(x, y)`	zur Stelle (x, y) springen, ohne zu zeichnen; die Richtung bleibt
`t.moveTo(x, y)`	zur Stelle (x, y) laufen; zeichnet, wenn der Stift unten ist
`t.home()`	zurück in die Mitte springen
`t.setPenColor(farbe)`	Stiftfarbe, z. B. `"red"` oder `pbc.RED`
`t.speed(s)`	Tempo: `1` langsam … `10` schnell, `0` sofort (Standard: `5`)
`t.hide()`, `t.show()`	das Dreieck verstecken / wieder zeigen
`t.clear()`, `t.clear(farbe)`	alles löschen, schwarz oder in einer Farbe

k16_strasse.py

```
1  import turtle
2
3  t = turtle.Turtle()
4  t.speed(0)
5  t.setDirection(90)
6  t.setPenColor("gray")
7  t.setPosition(-110, -30)        # Fahrbahnrand
8  t.forward(220)
9
10 t.setPenColor("white")
11 t.setPosition(-110, 0)          # Mittellinie
12 t.speed(4)
13 for i in range(11):
14     t.forward(12)       # Strich
15     t.penUp()
16     t.forward(8)        # Lücke
17     t.penDown()
```



- Den Fahrbahnrand (Zeile 6 bis 8) zeichnet die Turtle mit `speed(0)`, also sofort. Mit `setPosition` springt sie an den Anfang der Linie, ohne dabei zu zeichnen. Zeile 11 springt genauso zur Mittellinie.
- Die Mittellinie soll man entstehen sehen: Zeile 12 macht die Turtle langsamer.
- In der Schleife zeichnet sie einen Strich, hebt den Stift, läuft über die Lücke und senkt den Stift wieder. So entsteht eine gestrichelte Linie.

Farben: `setPenColor` versteht englische Namen in Anführungszeichen: `"red"`, `"green"`, `"blue"`, `"yellow"`, `"white"`, `"black"`, `"cyan"`, `"magenta"`, `"orange"`, `"purple"` und `"gray"`. Du kannst auch die Farbkonstanten aus `pbc` nehmen oder mit `pbc.rgb(r, g, b)` eine eigene Farbe mischen, z. B. `t.setPenColor(pbc.rgb(255, 105, 180))` für Rosa. Dann brauchst du zusätzlich `import pbc`.

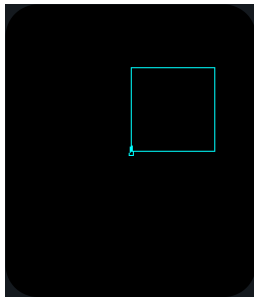
- Ein falscher Farbname ergibt **keine** Fehlermeldung: Mit `t.setPenColor("rot")` oder `"pink"` zeichnet die Turtle einfach weiß.
- Mische Turtle-Grafik nicht mit `pbc.clear`, `pbc.fill_rect` & Co. Die Turtle hat ihre eigene Zeichenfläche: Die erste Turtle macht den Bildschirm schwarz, und was du danach mit `pbc` zeichnest, kennt sie nicht. Es wird ganz oder teilweise übermalt, sobald sie sich bewegt. Den Hintergrund färbst du mit `t.clear(farbe)`.

16.4 Vielecke und Muster mit Schleifen und Funktionen

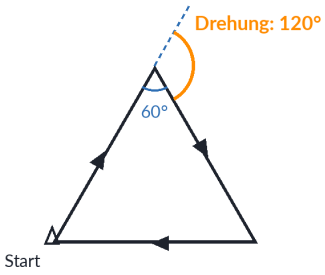
Ein Quadrat sind viermal „vorwärts, rechts drehen um 90 Grad“:

k16_quadrat.py

```
1 import turtle
2
3 t = turtle.Turtle()
4 t.setPenColor("cyan")
5 for i in range(4):
6     t.forward(80)
7     t.right(90)
```



Nach dem Quadrat steht die Turtle wieder am Start und schaut wieder nach oben. Sie hat sich viermal um 90 Grad gedreht, zusammen also um 360 Grad: einmal ganz herum. Das gilt für **jedes** Vieleck. Nach allen Ecken hat sich die Turtle einmal ganz herumgedreht. Bei n Ecken dreht sie sich an jeder Ecke also um 360 / n Grad (den **Außenwinkel**):



Die Turtle dreht sich an jeder Ecke um den Außenwinkel.

Nach allen Ecken hat sie sich einmal ganz herumgedreht: 360°.

Dreieck: 360 / 3 = 120°
Quadrat: 360 / 4 = 90°
n-Eck: 360 / n

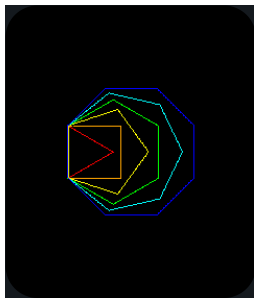
Daraus wird eine Funktion mit zwei Parametern:

k16_vielecke.py

```
1  import turtle
2
3  FARBEN = ["red", "orange", "yellow", "green", "cyan", "blue"]
4
5
6  def vieleck(t, ecken, seite):
7      winkel = 360 / ecken
8      for i in range(ecken):
9          t.forward(seite)
10         t.right(winkel)
```

k16_vielecke.py

```
13 t = turtle.Turtle()
14 t.hide()
15 t.speed(0)
16 t.setPosition(-60, -25)
17 for ecken in range(3, 9):
18     t.setPenColor(FARBEN[ecken - 3])
19     vieleck(t, ecken, 50)
```



- Die Funktion `vieleck` (Zeile 6 bis 10) bekommt die Turtle, die Zahl der Ecken und die Länge einer Seite. Weil die Turtle ein Parameter ist, kann die Funktion mit **jeder** Turtle zeichnen.
- Zeile 7 rechnet den Drehwinkel aus. `/` ergibt eine Kommazahl, z. B. 51,43 Grad beim Siebeneck. Mit `//` wäre es 51, und das Vieleck würde sich nicht schließen.
- Die Schleife in Zeile 17 zeichnet alle Vielecke vom Dreieck bis zum Achteck. Sie beginnen alle an derselben Stelle. Die Farbe holt Zeile 18 aus der Liste: Für 3 Ecken ist es `FARBEN[0]`.

Mit `speed(0)` (Zeile 15) erscheint jedes Vieleck sofort. Bei großen Figuren sparst du damit viel Wartezeit. Muster entstehen, wenn sich in jeder Runde etwas ändert: Dreh die Turtle nach jedem Vieleck ein Stück weiter, oder mach jede Linie etwas länger als die vorige. Probiere es aus!

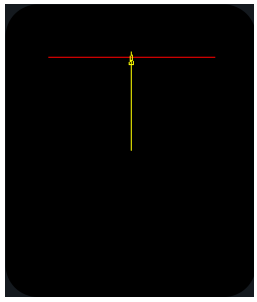
16.5 Die Turtle sieht nach vorn: `on_color`

`t.on_color(farbe)` prüft die Farbe des Pixels, das **3 Pixel vor** der Turtle liegt. Das Ergebnis ist `True`, wenn dort genau diese Farbe ist, sonst `False`. Die Turtle schaut nach vorn, damit sie nicht ihre eigene Spur oder ihr eigenes Dreieck sieht.

Die Hindernisse zeichnet eine zweite Turtle. Alle Turtles teilen sich eine Zeichenfläche. Nur die erste Turtle macht den Bildschirm schwarz, jede weitere zeichnet auf demselben Bild weiter.

k16_mauer.py

```
 1 import turtle
 2
 3 # Die erste Turtle baut eine rote Mauer.
 4 maurer = turtle.Turtle()
 5 maurer.hide()
 6 maurer.speed(0)
 7 maurer.setPenColor("red")
 8 maurer.setPosition(-80, 90)
 9 maurer.setDirection(90)
10 maurer.forward(160)
11
12 # Die zweite Turtle fährt los, bis sie die Mauer sieht.
13 flitzer = turtle.Turtle()
14 flitzer.setPenColor("yellow")
15 schritte = 0
16 while not flitzer.on_color("red"):
17     flitzer.forward(1)
18     schritte = schritte + 1
19 print(f"Mauer nach {schritte} Schritten erreicht")
```



Konsole

```
Mauer nach 87 Schritten erreicht
```

- Zeile 4 bis 10: Die erste Turtle heißt `maurer`. Sie ist versteckt und zeichnet sofort. Sie springt nach links oben, dreht sich nach rechts und zeichnet eine rote Linie, die Mauer.
- Zeile 13 legt die zweite Turtle `flitzer` an. Sie steht in der Mitte und schaut nach oben. Die Mauer bleibt stehen.
- Zeile 16: Solange vor dem Flitzer **nicht** Rot ist, geht er einen Pixel vorwärts und zählt den Schritt. Die Mauer liegt bei y = 90. Der Flitzer bleibt 3 Pixel davor stehen.

> - `on_color` vergleicht **genau**. Zeichne Hindernisse deshalb mit einem Farbnamen oder einer Farbkonstante und frage mit derselben Farbe ab: `on_color("red")` und `on_color(pbc.RED)` sind gleich.
> - Geh in kleinen Schritten (`forward(1)`). Mit `forward(5)` springt die Turtle über eine dünne Linie hinweg, ohne sie je zu sehen.

16.6 Selbstähnliche Figuren: Eine Funktion ruft sich selbst auf

Eine Funktion darf sich selbst aufrufen. Das nennt man **Rekursion**. Damit das nicht ewig weitergeht, braucht sie eine **Abbruchbedingung**: einen Fall, in dem sie sich nicht mehr selbst aufruft.

> Fehlt die Abbruchbedingung, ruft sich die Funktion immer wieder auf. Jeder Aufruf belegt etwas Speicher, bis Python mit einem `RuntimeError` abbricht (`maximum recursion depth exceeded`). Die Meldung ist sehr lang, weil jeder Aufruf eine eigene Zeile bekommt. Das Hinweisfenster von PBC+ Studio rät dann, eine Abbruchbedingung mit `if` einzubauen.

Ein Countdown mit Abbruchbedingung:

k16_countdown.py

```
1 def countdown(n):
2     if n == 0:              # Abbruchbedingung
3         print("Start!")
4         return
5     print(n)
6     countdown(n - 1)        # die Funktion ruft sich selbst auf
7
8
9 countdown(5)
```

Konsole

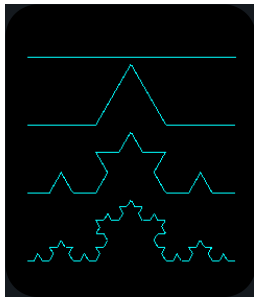
```
5
4
3
2
1
Start!
```

`countdown(5)` schreibt 5 und ruft `countdown(4)` auf. Das schreibt 4 und ruft `countdown(3)` auf, und so weiter. Bei `countdown(0)` greift die Abbruchbedingung in Zeile 2: „Start!“, und `return` beendet die Funktion, ohne dass sie sich noch einmal aufruft.

Mit Rekursion zeichnest du **selbstähnliche** Figuren: Figuren, deren Teile wie kleine Kopien des Ganzen aussehen. Die bekannteste ist die **Koch-Kurve**. Stufe 0 ist eine gerade Linie. Für die nächste Stufe wird jede Linie gedrittelt und das mittlere Drittel durch eine Spitze ersetzt:

k16_koch.py

```
 1 import turtle
 2
 3
 4 def koch(t, laenge, stufe):
 5     if stufe == 0:
 6         t.forward(laenge)
 7         return
 8     teil = laenge / 3
 9     koch(t, teil, stufe - 1)
10     t.left(60)
11     koch(t, teil, stufe - 1)
12     t.right(120)
13     koch(t, teil, stufe - 1)
14     t.left(60)
15     koch(t, teil, stufe - 1)
```



k16_koch.py

```
18 t = turtle.Turtle()
19 t.hide()
20 t.speed(0)
21 t.setPenColor("cyan")
22 y = 90
23 for stufe in range(4):
24     t.setPosition(-100, y)
25     t.setDirection(90)
26     koch(t, 200, stufe)
27     y = y - 65
```

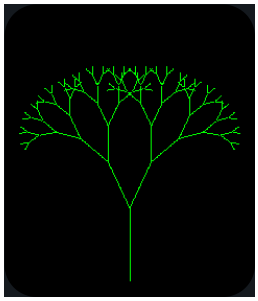
- Zeile 5 bis 7 ist die Abbruchbedingung: Bei Stufe 0 zeichnet die Turtle einfach eine Linie.
- Sonst zeichnet die Funktion vier Koch-Kurven der nächstkleineren Stufe, jede ein Drittel so lang (Zeile 9, 11, 13, 15). Dazwischen dreht sich die Turtle so, dass eine Spitze entsteht: links 60, rechts 120, links 60.
- Jede Stufe verringert die Zahl um 1, also landet jeder Aufruf irgendwann bei Stufe 0.
  Die Zahl der Linien wächst schnell (Stufe 3: 64, Stufe 5: 1024). Halte die Stufe klein und benutze `speed(0)`.

Ausprobieren: Ein Baum

Ein Baum ist auch selbstähnlich: Ein Ast teilt sich in zwei kleinere Äste, die sich wieder teilen, und so weiter.

k16_baum.py

```
 1 import turtle
 2
 3 WINKEL = 25
 4
 5
 6 def ast(t, laenge, stufe):
 7     if stufe == 0:
 8         return
 9     t.forward(laenge)
10     t.left(WINKEL)
11     ast(t, laenge * 0.7, stufe - 1)     # linker Teilbaum
12     t.right(2 * WINKEL)
13     ast(t, laenge * 0.7, stufe - 1)     # rechter Teilbaum
14     t.left(WINKEL)
15     t.penUp()
16     t.back(laenge)                      # zurück zum Anfang des Asts
17     t.penDown()
```



k16_baum.py

```
20 t = turtle.Turtle()
21 t.setPenColor("green")
22 t.speed(10)
23 t.penUp()
24 t.back(125)
25 t.penDown()
26 ast(t, 70, 7)
27 t.hide()
```

- Zeile 6: `ast` zeichnet einen Ast mit allem, was daran wächst. `stufe` sagt, wie oft er sich noch teilt. Bei Stufe 0 passiert nichts (Zeile 7 und 8).
- Zeile 9 bis 13: Die Turtle zeichnet den Ast, dreht sich nach links und zeichnet dort einen kleineren Baum (70 % so lang). Dann dreht sie sich nach rechts und zeichnet den zweiten kleinen Baum.
- Zeile 14 bis 17: Die Turtle dreht sich zurück und läuft mit gehobenem Stift rückwärts an den Anfang des Asts. Das ist wichtig: Jeder Aufruf von `ast` hinterlässt die Turtle genau dort, wo er sie bekommen hat. Nur so passt der nächste Ast an die richtige Stelle.

Zum Verändern: Probiere für `WINKEL` 15, 40 oder 90 und statt `0.7` die Werte `0.6` oder `0.8`. Färbe den Stamm braun (`pbc.rgb(150, 90, 40)`). Mit `random.randint(15, 35)` als Winkel wird jeder Baum anders.

Übungsaufgaben

Aufgabe 1 ⭐ Was zeichnet die Turtle? Welche Figur entsteht, wie groß ist sie, und wo liegt sie? Wo steht die Turtle am Ende, und wohin schaut sie? Erst denken, dann testen.

k16_aufgabe1.py

```
import turtle

t = turtle.Turtle()
t.setPenColor("orange")
for i in range(2):
    t.forward(100)
    t.right(90)
    t.forward(50)
    t.right(90)
```

Aufgabe 2 ⭐ Häuserreihe Schreibe eine Funktion `haus(t, groesse)` für ein Quadrat mit dreieckigem Dach. Die Turtle beginnt unten links mit Blick nach oben und steht am Ende wieder dort. Zeichne drei verschieden große Häuser nebeneinander.

Aufgabe 3 ⭐⭐ Finde die Fehler Statt eines grünen Siebenecks erscheint eine weiße Linie, die oben aus dem Bildschirm läuft, ohne Fehlermeldung. Finde die zwei Fehler.

k16_aufgabe3.py

```
import turtle


def vieleck(t, ecken, seite):
    winkel = 360 / ecken
    for i in range(ecken):
        t.forward(seite)
    t.right(winkel)


t = turtle.Turtle()
t.setPenColor("gruen")
vieleck(t, 7, 45)
```

Aufgabe 4 ⭐⭐ Olympische Ringe Ein 36-Eck mit kurzen Seiten sieht aus wie ein Kreis. Schreibe `ring(t, x, y, farbe)` und zeichne die fünf Ringe aus einer Liste von Listen `[x, y, farbe]` (Weiß statt Schwarz).

Aufgabe 5 ⭐⭐ Wie weit bis zur Wand? Eine zweite Turtle zeichnet ein rotes Rechteck um die Mitte (nicht genau mittig). Schreibe `abstand(t)`: Die Turtle läuft mit gehobenem Stift bis zur Wand, zählt die Schritte, läuft zurück und gibt die Zahl zurück. Gib die Abstände in alle vier Richtungen aus.

Aufgabe 6 ⭐⭐⭐ Schneeflocke Setze drei Koch-Kurven der Stufe 3 so zusammen, dass eine Schneeflocke entsteht. Überlege zuerst: Um wie viel Grad muss sich die Turtle nach jeder Kurve drehen, wenn die drei Kurven ein Dreieck bilden sollen?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `import turtle` und `t = turtle.Turtle()` legen eine Turtle an. Sie steht in der Mitte bei (0, 0) und schaut nach oben; y wächst nach **oben**.
- Bewegen: `forward`, `back`, `left`, `right`; Richtung direkt: `setDirection(w)` mit 0 = oben, 90 = rechts, 180 = unten, 270 = links; springen ohne Linie: `setPosition(x, y)`, laufen mit Linie: `moveTo(x, y)`.
- Stift und Tempo: `penUp`, `penDown`, `setPenColor("red")`, `speed(0)` für sofort, `hide` / `show`.
- Ein Vieleck mit n Ecken: n-mal vorwärts und um 360 / n Grad drehen.
- `on_color(farbe)` prüft das Pixel 3 Pixel vor der Turtle. Hindernisse in genauer Farbe zeichnen, in kleinen Schritten fahren.
- Rekursion: Eine Funktion ruft sich selbst auf. Sie braucht immer eine Abbruchbedingung.

KAPITEL 17

Der Canvas – flimmerfrei zeichnen

Worum geht's? Bisher hast du direkt auf den Bildschirm gezeichnet. Jede Linie war sofort zu sehen, und bei Animationen hat es geflackert. In diesem Kapitel malst du ein Bild zuerst unsichtbar im Speicher und zeigst es dann auf einen Schlag. Außerdem lädst du fertige Bilder (PNG-Dateien) und setzt sie zu einer Szene zusammen.

17.1 Ein Bild im Speicher: `pbc.Canvas`

Ein **Canvas** (englisch für Leinwand) ist ein Bild, das nur im Speicher des PicoBoy liegt. Du kannst darauf zeichnen wie auf dem Bildschirm, aber niemand sieht es, bis du es mit `pbc.show` auf den Bildschirm kopierst.

k17_speicherbild.py

```
1  import pbc
2  import time
3
4  pbc.clear()
5
6  bild = pbc.Canvas(240, 280)
7  bild.fill(pbc.BLUE)
8  bild.text24("Fertig!", 82, 128, pbc.WHITE)
9  print("Das Bild ist im Speicher fertig.")
10
11 time.sleep(2)
12 pbc.show(bild)
13 print("Jetzt ist es auf dem Bildschirm.")
```



Terminal

```
Das Bild ist im Speicher fertig.
Jetzt ist es auf dem Bildschirm.
```

- Zeile 6 legt einen Canvas an, der 240 Pixel breit und 280 Pixel hoch ist, also genau so groß wie der Bildschirm. `pbc.Canvas(...)` liefert ein **Objekt**: ein Ding, das Werte speichert und eigene Befehle hat. Wir merken es uns in der Variablen `bild`.
- Zeile 7 und 8 malen auf den Canvas. Die Befehle heißen fast wie die bekannten `pbc`-Befehle, nur steht vorn `bild.` statt `pbc.`: Der Befehl gehört zu diesem Canvas. Solche Befehle eines Objekts heißen **Methoden**.
- Nach Zeile 9 ist das Bild fertig, aber der Bildschirm bleibt noch zwei Sekunden schwarz.
- Erst `pbc.show(bild)` in Zeile 12 kopiert das ganze Bild auf einmal auf den Bildschirm.

Ein Canvas braucht Speicher: 2 Byte pro Pixel, bei 240 × 280 Pixeln also 134 400 Byte. Lege einen großen Canvas deshalb **einmal** am Anfang an und benutze ihn immer wieder.

17.2 Zeichnen auf dem Canvas, anzeigen mit `pbc.show`

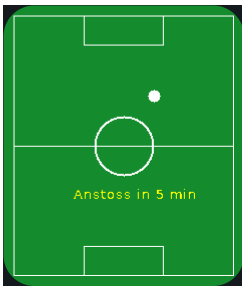
Auf einem Canvas gibt es fast alle Zeichenbefehle, die du aus Kapitel 5 kennst. Ist `c` ein Canvas, dann gilt:

Auf dem Canvas	Bedeutung
`c.fill(farbe)`	ganzen Canvas füllen (statt `pbc.clear`)
`c.pixel(x, y, farbe)`, `c.line(x1, y1, x2, y2, farbe)`	Pixel, Linie (auch `hline`, `vline`)
`c.rect(x, y, b, h, farbe)`, `c.fill_rect(x, y, b, h, farbe)`	Rechteck als Umriss oder gefüllt
`c.ellipse(mx, my, rx, ry, farbe)`, `…, farbe, True)`	Ellipse als Umriss, gefüllt
`c.text(s, x, y, farbe)`, `c.text12/18/24(s, x, y, farbe)`	Text
`c.text12_width(s)` (ebenso 18, 24), `c.width`, `c.height`	Breite eines Texts; Größe des Canvas

Die Koordinaten zählen auf dem Canvas genauso wie auf dem Bildschirm: (0, 0) ist links oben.

k17_spielfeld.py

```
 1 import pbc
 2
 3 feld = pbc.Canvas(240, 280)
 4 feld.fill(pbc.rgb(30, 140, 50))
 5
 6 # Linien: Rand, Mittellinie, Mittelkreis (weisser Kreis,
 7 # darin ein etwas kleinerer gruener)
 8 feld.rect(10, 10, 220, 260, pbc.WHITE)
 9 feld.ellipse(120, 140, 30, 30, pbc.WHITE, True)
10 feld.ellipse(120, 140, 28, 28, pbc.rgb(30, 140, 50), True)
11 feld.hline(10, 140, 220, pbc.WHITE)
12
13 # Tore oben und unten
14 feld.rect(80, 10, 80, 30, pbc.WHITE)
15 feld.rect(80, 240, 80, 30, pbc.WHITE)
16
17 # Ball: gefuellter Kreis, True steht fuer "gefuellt"
18 feld.ellipse(150, 90, 6, 6, pbc.WHITE, True)
19 feld.text12("Anstoss in 5 min", 70, 180, pbc.YELLOW)
20
21 pbc.show(feld)
```



- Zeile 4 füllt den Canvas mit Rasengrün.
- Zeile 9 und 10 zeichnen den Mittelkreis: erst einen weißen gefüllten Kreis, dann einen etwas kleineren grünen darüber. Übrig bleibt ein weißer Ring.
- In Zeile 18 steht als sechstes Argument `True`. Damit wird die Ellipse gefüllt.
- Erst Zeile 21 zeigt das fertige Spielfeld.

Zwei Unterschiede zum direkten Zeichnen: Einen Canvas löschst du mit `c.fill(farbe)`. Einen Befehl `clear` hat er nicht. `bild.clear()` in Zeile 4 eines Programms bricht so ab:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
AttributeError: 'Canvas' object has no attribute 'clear'
```

Für gefüllte Formen schreibst du auf dem Canvas nur `True`, ohne `fill=`: `c.ellipse(120, 140, 30, 30, pbc.WHITE, True)`. Mit `fill=True` bricht das Programm ab:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: function doesn't take keyword arguments
```

Kleine Canvas an beliebiger Stelle. Ein Canvas muss nicht so groß wie der Bildschirm sein. `pbc.show(c, x, y)` setzt seine linke obere Ecke an die Stelle (x, y). Nur dieser Teil des Bildschirms ändert sich, der Rest bleibt stehen:

k17_anzeige.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear(pbc.BLUE)
 5 pbc.text18("Punktestand", 70, 60, pbc.WHITE)
 6
 7 tafel = pbc.Canvas(120, 40)
 8 for punkte in range(0, 101, 5):
 9     tafel.fill(pbc.BLACK)
10     tafel.rect(0, 0, 120, 40, pbc.YELLOW)
11     tafel.text24(str(punkte), 45, 8, pbc.YELLOW)
12     pbc.show(tafel, 60, 120)
13     time.sleep_ms(100)
```



- Zeile 4 und 5 zeichnen direkt auf den Bildschirm. Das geschieht nur einmal.
- Zeile 7 legt eine kleine Anzeigetafel von 120 × 40 Pixeln an.
- In der Schleife wird die Tafel jedes Mal gelöscht, neu beschriftet und in Zeile 12 an die Stelle (60, 120) kopiert. Die Überschrift bleibt dabei erhalten. Die Zahl wechselt, ohne zu flackern.

Der Canvas muss ganz auf den Bildschirm passen: x + Breite höchstens 240, y + Höhe höchstens 280, und x und y nicht negativ. Auf dem Gerät bricht das Programm sonst mit `ValueError: blit region out of bounds` ab. Der Emulator schneidet den überstehenden Teil einfach ab, du merkst den Fehler dort also nicht.

17.3 Bilder laden: `pbc.load_image`

Ein Bild mit vielen Details selbst zu zeichnen ist mühsam. Fertige Bilder im Format **PNG** kann der PicoBoy laden. Die Bilder zu diesem Buch, zum Beispiel `meer.png` und `fisch.png`, liegen bei den Beispielprogrammen im Ordner `material`. Du kannst auch eigene PNG-Bilder benutzen.

Bilder hochladen. Bevor ein Programm ein Bild laden kann, muss das Bild dort liegen, wo das Programm läuft: im Emulator oder auf dem Gerät. Wähle dazu das Ziel (**Emulator** oder **PBC+**), dann **Dateien → Bilder verwalten…** und **PNG hochladen…**. Wie das Schritt für Schritt geht, steht mit Bildern in Abschnitt 18.1. Lade für dieses Kapitel `meer.png` und `fisch.png` hoch.

Ein Bild laden und zeigen. `pbc.load_image("name.png")` lädt ein Bild und liefert es als Canvas. Mit allem, was du über den Canvas weißt, kannst du es also anzeigen, darauf zeichnen und es mit anderen Bildern kombinieren:

k17_bild_laden.py

```
 1 import pbc
 2 import time
 3
 4 meer = pbc.load_image("meer.png")
 5 fisch = pbc.load_image("fisch.png")
 6 print("Meer:", meer.width, "x", meer.height)
 7 print("Fisch:", fisch.width, "x", fisch.height)
 8
 9 # Direkt auf den Bildschirm
10 pbc.show(meer)
11 pbc.show(fisch, 40, 60)
12 pbc.show(fisch, 150, 200)
13 time.sleep(2)
14
15 # Auf das Meer-Bild stempeln, dann zeigen
16 fisch.stamp_on(meer, 150, 120)
17 fisch.stamp_on(meer, 100, 170)
18 meer.text18("Im Meer", 80, 20, pbc.WHITE)
19 pbc.show(meer)
```

Terminal

```
Meer: 240 x 280
Fisch: 24 x 16
```

- Zeile 4 und 5 laden zwei Bilder. Zeile 6 und 7 zeigen, dass jedes Bild seine eigene Breite und Höhe hat.
- Zeile 10 zeigt das Meer-Bild, Zeile 11 und 12 zeigen den Fisch an zwei Stellen. Um den Fisch herum ist das Bild **durchsichtig**: Dort bleibt das Meer sichtbar.

- `fisch.stamp_on(meer, 150, 120)` in Zeile 16 **stempelt** den Fisch auf den Canvas `meer`. Auch dabei bleiben die durchsichtigen Pixel durchsichtig. Danach ist der Fisch ein Teil des Meer-Bildes.
- Zeile 18 schreibt eine Überschrift auf das Bild, Zeile 19 zeigt es.



Achte auf das Ergebnis: Die beiden Fische aus Zeile 11 und 12 sind am Ende verschwunden. Sie waren nur auf dem Bildschirm, nicht im Canvas `meer`, und `pbc.show(meer)` hat sie übermalt.

> **Durchsichtig** sind die Pixel, die im PNG als durchsichtig gespeichert sind. Jedes Malprogramm kann solche PNGs speichern (oft heißt die Einstellung „Transparenz“ oder „Alpha-Kanal“). Ein Canvas, den du selbst mit `pbc.Canvas` anlegst, hat keine durchsichtigen Pixel.

Ist das Bild nicht hochgeladen oder der Name falsch geschrieben, bricht das Programm ab. Groß- und Kleinschreibung zählen auch bei Dateinamen. Hier steht in Zeile 3 `pbc.load_image("Meer.png")` statt `"meer.png"`:

Terminal
```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
OSError: [Errno 2] ENOENT
```

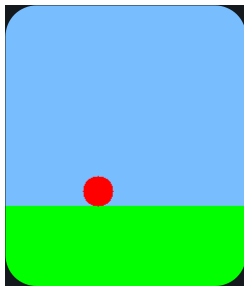
`ENOENT` bedeutet „Datei nicht gefunden“. Das Hinweisfenster von PBC+ Studio erinnert dich daran, das Bild über **Dateien → Bilder verwalten…** hochzuladen.

17.4 Animation ohne Flackern

In Kapitel 6 hat deine Animation geflackert. Das Programm `k17_flackern.py` zeigt es noch einmal: Ein Ball rollt über eine Wiese, und in jedem Schleifendurchlauf malt `pbc.clear(...)` zuerst den Himmel über den ganzen Bildschirm, also auch über den Ball. Einen Moment lang ist der Bildschirm ohne Ball zu sehen, erst danach kommen Wiese und Ball wieder hin. Dieses ständige Verschwinden und Wiederauftauchen sieht dein Auge als Flackern.

Mit einem Canvas siehst du nur fertige Bilder:

k17_ohne_flackern.py
```
1 import pbc
2 import time
3
4 bild = pbc.Canvas(240, 280)
5
6 x = 0
7 while x < 240:
8     # 1. Ganzes Bild im Speicher neu zeichnen
9     bild.fill(pbc.rgb(120, 190, 255))
```



```
10      bild.fill_rect(0, 200, 240, 80, pbc.GREEN)
11      bild.ellipse(x, 185, 15, 15, pbc.RED, True)
12      # 2. Fertiges Bild auf einmal zeigen
13      pbc.show(bild)
14      # 3. Kurz warten
15      x = x + 4
16      time.sleep_ms(20)
```

Die Schleife hat jetzt drei feste Schritte:

1. **Alles im Speicher neu zeichnen** (Zeile 9 bis 11): Himmel, Wiese und Ball an der neuen Stelle.
2. **Das fertige Bild zeigen** (Zeile 13): `pbc.show(bild)` ersetzt das alte Bild auf einen Schlag. Ein halbfertiges Bild ist nie zu sehen.
3. **Warten** (Zeile 16), dann geht es von vorn los.

Der Canvas wird in Zeile 4 nur einmal angelegt, **vor** der Schleife.

Für jedes Bild zeichnest du alles neu, auch was sich nicht bewegt. Im Speicher geht das schnell. Am längsten dauert `pbc.show` (etwa 40 ms für den ganzen Bildschirm), mehr als etwa 20 Bilder pro Sekunde schaffst du also nicht.

> Wenn du einen Canvas benutzt, zeichne **nicht** zusätzlich direkt mit `pbc.text12`, `pbc.clear` & Co. auf den Bildschirm. Das nächste `pbc.show(bild)` übermalt alles, was nicht im Canvas steht. Schreibe deshalb auch Texte wie den Punktestand in den Canvas.

Ausprobieren: Der Wetterbericht

Der Wetterbericht zeigt für mehrere Tage das Wetter mit einem kleinen, bewegten Bild. Mit dem Joystick blätterst du zwischen den Tagen.

k17_wetter.py

```
 1 import pbc
 2 import time
 3
 4 # Tag, Wetter, hoechste und tiefste Temperatur
 5 TAGE = [["Montag", "Sonne", 24, 13],
 6         ["Dienstag", "Wolken", 19, 11],
 7         ["Mittwoch", "Regen", 15, 9]]
 8
 9 bild = pbc.Canvas(240, 280)
10
11
12 def zeichne_wolke(x, y):
13     bild.ellipse(x, y, 30, 18, pbc.WHITE, True)
14     bild.ellipse(x - 26, y + 8, 22, 12, pbc.WHITE, True)
15     bild.ellipse(x + 26, y + 8, 22, 12, pbc.WHITE, True)
```

```
18 def zeichne_tag(tag, schritt):
19     name, wetter, hoch, tief = tag
20     if wetter == "Sonne":
21         bild.fill(pbc.rgb(70, 150, 255))
22         bild.ellipse(120, 110, 35, 35, pbc.YELLOW, True)
23     elif wetter == "Wolken":
24         bild.fill(pbc.rgb(120, 160, 210))
25         bild.ellipse(150, 95, 28, 28, pbc.YELLOW, True)
26         # Die Wolke zieht langsam von links nach rechts.
27         zeichne_wolke(schritt * 3 % 300 - 30, 120)
28     else:
29         bild.fill(pbc.GRAY)
30         zeichne_wolke(120, 90)
31         for i in range(8):
32             x = 75 + i * 13
33             y = 115 + (schritt * 6 + i * 23) % 60
34             bild.line(x, y, x - 3, y + 10, pbc.CYAN)
35     bild.text24(name, 20, 15, pbc.WHITE)
36     bild.text18(f"{hoch} / {tief} Grad", 20, 195, pbc.WHITE)
37     bild.text12("links/rechts: anderer Tag", 20, 245, pbc.WHITE)
38     pbc.show(bild)
39
40
41 nummer = 0
42 schritt = 0
43 while True:
44     if pbc.was_pressed_right():
45         nummer = (nummer + 1) % len(TAGE)
46     if pbc.was_pressed_left():
47         nummer = (nummer - 1) % len(TAGE)
48     zeichne_tag(TAGE[nummer], schritt)
49     schritt = schritt + 1
50     time.sleep_ms(30)
```





- Zeile 5 bis 7: Jeder Tag ist eine Liste mit Name, Wetter und zwei Temperaturen, zusammen eine Liste von Listen (Kapitel 14). Zeile 9 legt **einen** Canvas für das ganze Programm an.
- `zeichne_tag` zeichnet das komplette Bild für einen Tag. Zeile 19 holt die vier Werte aus der Liste `tag` in vier Variablen. Erst ganz am Ende, in Zeile 38, kommt `pbc.show(bild)`.
- Der Parameter `schritt` zählt die Bilder der Animation. Bei „Wolken“ wandert die Wolke mit jedem Schritt 3 Pixel weiter (Zeile 27); der Rest `% 300` lässt sie nach 300 Pixeln wieder links anfangen. Bei „Regen“ hat jeder der acht Tropfen wegen `i * 23` eine andere Höhe, und `% 60` lässt ihn oben wieder anfangen, wenn er 60 Pixel gefallen ist.
- Die Hauptschleife (ab Zeile 43) ist eine Spielschleife wie in Kapitel 8. `(nummer - 1) % len(TAGE)` macht aus −1 die letzte Nummer: Links von Montag kommt Mittwoch.

Zum Verändern: Füge Tage hinzu. Lass die Sonne pulsieren oder bei Regen eine Pfütze wachsen.

Übungsaufgaben

Aufgabe 1 ⭐ **Was erscheint?** Wie viele Quadrate siehst du am Ende, wo stehen sie, wie sehen sie aus? Überlege erst, prüfe dann im Emulator.

k17_aufgabe1.py

```
import pbc

pbc.clear()
kachel = pbc.Canvas(60, 60)
kachel.fill(pbc.RED)
kachel.ellipse(30, 30, 20, 20, pbc.YELLOW, True)
pbc.show(kachel, 0, 0)
pbc.show(kachel, 180, 0)
kachel.fill(pbc.BLUE)
pbc.show(kachel, 90, 110)
```

Aufgabe 2 ⭐ **Finde die Fehler** Das Programm soll eine schwarz-rot-goldene Flagge zeigen. Es enthält drei Fehler. Nur einer davon erzeugt eine Fehlermeldung. Finde alle drei.

k17_aufgabe2.py

```
import pbc

pbc.clear(pbc.GRAY)
flagge = pbc.Canvas(240, 150)
flagge.clear(pbc.BLACK)
pbc.fill_rect(0, 50, 240, 50, pbc.RED)
flagge.fill_rect(0, 100, 240, 50, pbc.rgb(255, 200, 0))
```

Aufgabe 3 ⭐⭐ **Gewitter** Erweitere den Wetterbericht um einen vierten Tag „Donnerstag“ mit dem Wetter „Gewitter“: dunkler Himmel, eine Wolke und ein gelber Blitz aus drei Linien. Ab und zu soll der ganze Himmel kurz weiß aufleuchten.

Aufgabe 4 ⭐⭐ **Countdown** Schreibe einen Countdown von 10 bis 0. Die Zahl steht groß in der Mitte, darunter ist ein grüner Balken, der mit jeder Sekunde kürzer wird. Am Ende erscheint „Start!“. Benutze einen Canvas. Zum Zentrieren hilft `text24_width`.

Aufgabe 5 ⭐⭐⭐ Laufschrift Ein Text läuft in einem 40 Pixel hohen Band von rechts nach links über den Bildschirm. Ist er links ganz verschwunden, kommt er rechts wieder herein. Mit dem Joystick hoch und runter wird die Laufschrift schneller und langsamer. Tipp: Benutze einen Canvas von 240 × 40 Pixeln. Ein Text darf auch bei einem negativen x anfangen, dann ist nur sein rechter Teil zu sehen.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `c = pbc.Canvas(breite, hoehe)` legt ein Bild im Speicher an. Darauf zeichnest du mit `c.fill`, `c.line`, `c.fill_rect`, `c.ellipse`, `c.text24` … Gefüllte Formen: `True` als letztes Argument.
- `pbc.show(c)` kopiert den Canvas auf den Bildschirm, `pbc.show(c, x, y)` an die Stelle (x, y). Er muss ganz auf den Bildschirm passen.
- `pbc.load_image("name.png")` lädt ein Bild als Canvas. Vorher muss es über **Dateien → Bilder verwalten…** hochgeladen sein.
- `bild.stamp_on(ziel, x, y)` stempelt ein Bild mit seinen durchsichtigen Stellen auf einen anderen Canvas.
- Flimmerfreie Animation: Canvas einmal anlegen; in der Schleife alles neu in den Canvas zeichnen, `pbc.show`, warten.
- Was nur direkt auf dem Bildschirm steht, übermalt das nächste `pbc.show`.

KAPITEL 18

Sprites – Figuren, die sich bewegen

Worum geht's? Ein **Sprite** ist eine Figur aus einem Bild, die du über den Bildschirm bewegen kannst: eine Rakete, ein Fisch, eine Spielfigur. Um das Neuzeichnen musst du dich dabei nicht kümmern, das erledigt der PicoBoy für dich. In diesem Kapitel lernst du, Sprites anzulegen, zu zeigen, zu bewegen und vor einen Hintergrund zu setzen.

18.1 Ein Sprite anlegen: `pbc.sprite("bild.png")`

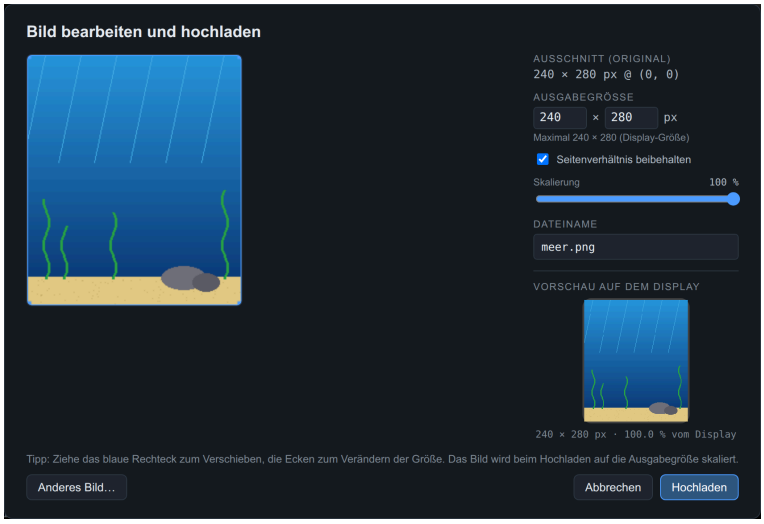
Jedes Sprite braucht ein Bild im PNG-Format. Die Bilder zu diesem Kapitel (`rakete.png`, `stern.png`, `ufo.png`, `geist.png`, `weltraum.png`, `meer.png`, `fisch.png`, `hai.png`, `krabbe.png`, `kugel.png`) liegen bei den Beispielprogrammen im Ordner `material`.

Bilder hochladen. Ein Programm findet nur Bilder, die vorher hochgeladen wurden, und zwar dorthin, wo es läuft:

1. Wähle oben rechts das Ziel: **Emulator** oder **PBC+**. Für das Gerät klickst du vorher auf **Verbinden**.
2. Wähle **Dateien → Bilder verwalten…**. Das Fenster zeigt alle Bilder, die schon im Emulator (oder auf dem Gerät) sind.



3. Klicke auf **PNG hochladen…** und dann auf **Bild von der Festplatte wählen…**. Wähle die Datei aus.
4. Das Fenster **Bild bearbeiten und hochladen** zeigt das Bild, seine Ausgabegröße und den Dateinamen. Lass beides unverändert (beim Verkleinern werden Farben gemischt) und klicke auf **Hochladen**. Wiederhole das für jedes Bild. Mehrere Bilder auf einmal gehen mit **ZIP hochladen…**: Alle PNG-Bilder aus der ZIP-Datei werden unverändert hochgeladen, ohne dieses Fenster.

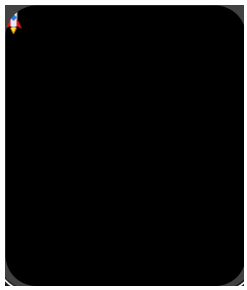


Für den **Emulator** werden die Bilder im Browser gespeichert. Sie bleiben dort, auch wenn du die Seite neu lädst, aber nur in diesem Browser auf diesem Computer. Für das **Gerät** lädst du die Bilder mit dem Ziel **PBC+** noch einmal hoch; sie liegen dann im Speicher des PicoBoy und bleiben auch nach dem Ausschalten erhalten.

Das erste Sprite.

k18_erstes_sprite.py

```
1 import pbc
2 import time
3
4 rakete = pbc.sprite("rakete.png")
5 print("Position:", rakete.x, rakete.y)
6 time.sleep(2)
7 rakete.show()
8 print("Jetzt ist sie zu sehen.")
```



Terminal

```
Position: 0 0
Jetzt ist sie zu sehen.
```

- Zeile 4 lädt das Bild `rakete.png` und macht daraus ein Sprite. Wie ein Canvas ist ein Sprite ein Objekt; wir merken es uns in der Variablen `rakete`.
- Ein neues Sprite steht an der Position (0, 0), also links oben, und ist **unsichtbar**. Zwei Sekunden lang passiert deshalb nichts.
- Erst `rakete.show()` in Zeile 7 macht es sichtbar.

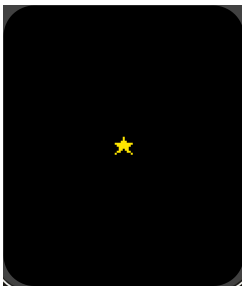
Die Position eines Sprites ist die **linke obere Ecke** seines Bildes, genau wie bei Rechtecken und Texten.

18.2 Zeigen und verstecken

`show()` macht ein Sprite sichtbar, `hide()` versteckt es wieder. Das Sprite behält dabei seine Position.

k18_blinken.py

```
 1 import pbc
 2 import time
 3
 4 stern = pbc.sprite("stern.png")
 5 stern.set_to(111, 131)
 6 for i in range(5):
 7     stern.show()
 8     time.sleep_ms(400)
 9     stern.hide()
10     time.sleep_ms(400)
11 stern.show()
```



- Zeile 5 setzt den Stern in die Mitte, noch unsichtbar.
- Die Schleife zeigt und versteckt ihn fünfmal. Danach bleibt er sichtbar (Zeile 11).

Bei jedem `show()` und `hide()` zeichnet der PicoBoy den **ganzen Bildschirm neu**: zuerst den Hintergrund (ohne eigenen Hintergrund ist er schwarz), dann alle sichtbaren Sprites. Ein Sprite, das später gezeigt wurde, liegt über den früheren. Deshalb flackert es nicht: Intern benutzt der PicoBoy dafür einen Canvas, wie du ihn aus Kapitel 17 kennst.

18.3 Setzen, versetzen, gleiten

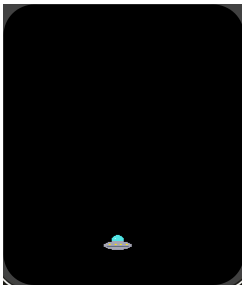
Für Bewegungen gibt es vier Befehle:

Befehl	Bedeutung
`s.set_to(x, y)`	sofort an die Stelle (x, y) **setzen**
`s.set_by(dx, dy)`	sofort um dx nach rechts und dy nach unten **versetzen**
`s.glide_to(x, y, ms)`	in `ms` Millisekunden zur Stelle (x, y) **gleiten**
`s.glide_by(dx, dy, ms)`	in `ms` Millisekunden um dx, dy gleiten

Negative Werte bei `set_by` und `glide_by` bewegen nach links oder oben.

k18_ufo.py

```
1 import pbc
2 import time
3
4 ufo = pbc.sprite("ufo.png")
5 ufo.set_to(10, 30)
6 ufo.show()
```



```python
# Gleiten: 2 Sekunden nach rechts unten
ufo.glide_to(200, 130, 2000)
print("angekommen bei", ufo.x, ufo.y)

# Gleiten um eine Strecke: 100 Pixel nach links
ufo.glide_by(-100, 0, 1000)
print("jetzt bei", ufo.x, ufo.y)

# Versetzen in kleinen Schritten: Das Ufo sinkt.
for i in range(20):
    ufo.set_by(0, 5)
    time.sleep_ms(50)
print("gelandet bei", ufo.x, ufo.y)
```

Terminal

```
angekommen bei 200 130
jetzt bei 100 130
gelandet bei 100 230
```

- Zeile 9 lässt das Ufo zwei Sekunden lang nach rechts unten gleiten. `glide_to` **wartet**, bis das Ufo angekommen ist. Erst dann geht es mit Zeile 10 weiter. Deshalb steht in der Ausgabe schon die Zielposition.
- Zeile 13 gleitet um 100 Pixel nach links, ganz gleich, wo das Ufo gerade ist.
- Zeile 17 bis 19 versetzen das Ufo zwanzigmal um 5 Pixel nach unten. `set_by` wartet nicht; die Pause macht `time.sleep_ms` in Zeile 19.

`glide_to` ist praktisch für Bewegungen, die genau so ablaufen sollen, wie du sie geplant hast. Während das Sprite gleitet, kann dein Programm aber nichts anderes tun, zum Beispiel keine Tasten abfragen. Für Spiele nimmst du deshalb `set_by` in einer Spielschleife.

18.4 Position und Größe abfragen

Ein Sprite kennt seine Position: `s.x` und `s.y` sind die Koordinaten der linken oberen Ecke. `s.width` und `s.height` sind Breite und Höhe des Bildes in Pixeln. Damit kannst du zum Beispiel verhindern, dass eine Figur den Bildschirm verlässt:

k18_steuern.py

```python
import pbc
import time

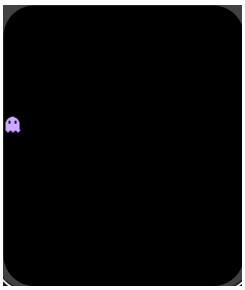
SCHRITT = 3

geist = pbc.sprite("geist.png")
geist.set_to(112, 132)
geist.show()
print("Breite:", geist.width, "Hoehe:", geist.height)
```

```
11 while True:
12     if pbc.pressed_left() and geist.x >= SCHRITT:
13         geist.set_by(-SCHRITT, 0)
14     if pbc.pressed_right() and geist.x + geist.width + SCHRITT <= pbc.WIDTH:
15         geist.set_by(SCHRITT, 0)
16     if pbc.pressed_up() and geist.y >= SCHRITT:
17         geist.set_by(0, -SCHRITT)
18     if pbc.pressed_down() and geist.y + geist.height + SCHRITT <= pbc.HEIGHT:
19         geist.set_by(0, SCHRITT)
20     if pbc.was_pressed_a():
21         print("Geist bei", geist.x, geist.y)
22     time.sleep_ms(20)
```

Terminal

```
Breite: 16 Hoehe: 16
Geist bei 142 111
Geist bei 1 111
```



- Zeile 12: Nach links darf der Geist nur, wenn danach x noch mindestens 0 ist, wenn also gerade `geist.x >= SCHRITT` gilt.
- Zeile 14: Nach rechts darf er nur, wenn auch sein **rechter** Rand danach noch im Bildschirm ist. Der rechte Rand liegt bei `geist.x + geist.width`.
- Zeile 16 und 18 machen dasselbe für oben und unten.
- Mit Taste A schreibt Zeile 21 die Position ins Terminal. Im Beispiel wurde der Geist erst nach rechts oben und dann ganz nach links gesteuert. Weil er immer 3 Pixel springt, bleibt er bei x = 1 stehen.

> `s.x` und `s.y` kannst du lesen, aber **verschiebe ein Sprite nicht** mit `s.x = 100`. Die Zahl ändert sich zwar, auf dem Bildschirm bleibt das Sprite aber stehen, bis es irgendwann neu gezeichnet wird. Benutze immer `set_to`, `set_by`, `glide_to` oder `glide_by`.

18.5 Sprites vor einem Hintergrund

Ohne eigenen Hintergrund ist hinter den Sprites alles schwarz. Einen Hintergrund legst du mit einem Canvas fest, zum Beispiel einem geladenen Bild:

```
weltraum = pbc.load_image("weltraum.png")
pbc.Sprite._background = weltraum
pbc.show(weltraum)
```

`pbc.Sprite._background` ist die Stelle, an der sich alle Sprites ihren Hintergrund merken. Ab jetzt wird bei jedem Neuzeichnen zuerst `weltraum` gezeichnet, dann die Sprites darüber. `pbc.show(weltraum)` zeigt den Hintergrund (mit allen sichtbaren Sprites darauf) sofort an.

Texte vor dem Hintergrund. Was du direkt auf den Bildschirm zeichnest, überlebt das nächste Neuzeichnen der Sprites nicht:

k18_text_weg.py

```
 1 import pbc
 2 import time
 3
 4 weltraum = pbc.load_image("weltraum.png")
 5 pbc.Sprite._background = weltraum
 6 pbc.show(weltraum)
 7
 8 rakete = pbc.sprite("rakete.png")
 9 rakete.set_to(112, 230)
10 rakete.show()
11
12 pbc.text18("Start!", 90, 120, pbc.YELLOW)
13 time.sleep_ms(2000)
14 rakete.glide_by(0, -200, 2000)
```



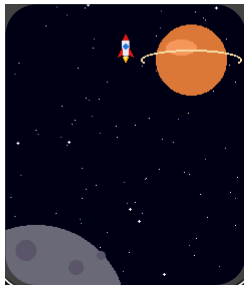
vor dem Start

Zeile 12 schreibt „Start!“ direkt auf den Bildschirm. Sobald die Rakete in Zeile 14 losfliegt, zeichnet der PicoBoy den Bildschirm aus Hintergrund und Sprites neu, und der Text ist verschwunden.

Schreibe Texte deshalb **in den Hintergrund-Canvas** und zeige ihn danach mit `pbc.show`:

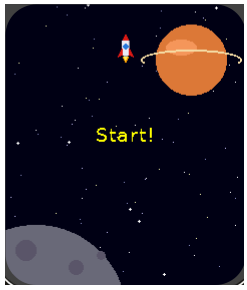
k18_text_bleibt.py

```
 1 import pbc
 2 import time
 3
 4 weltraum = pbc.load_image("weltraum.png")
 5 pbc.Sprite._background = weltraum
 6
 7 rakete = pbc.sprite("rakete.png")
 8 rakete.set_to(112, 230)
 9 rakete.show()
10
11 weltraum.text18("Start!", 90, 120, pbc.YELLOW)
12 pbc.show(weltraum)
13 time.sleep_ms(2000)
14 rakete.glide_by(0, -200, 2000)
```



nach dem Start

Zeile 11 schreibt in den Canvas `weltraum`, Zeile 12 zeigt ihn. Weil der Text jetzt zum Hintergrund gehört, wird er bei jedem Neuzeichnen mitgezeichnet.



> Soll sich ein Text ändern, zum Beispiel ein Punktestand, dann male vorher ein gefülltes Rechteck über den alten Text: `weltraum.fill_rect(…)`, dann `weltraum.text18(…)`, dann `pbc.show(weltraum)`.

Ausprobieren: Das Aquarium

Im Aquarium schwimmen vier Fische und ein Hai von links nach rechts, und eine Krabbe läuft über den Sand.

k18_aquarium.py

```
import pbc
import random
import time

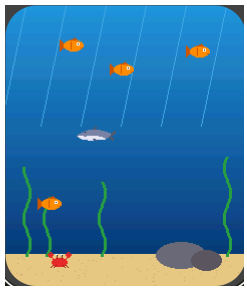
meer = pbc.load_image("meer.png")
pbc.Sprite._background = meer
pbc.show(meer)

# Fische: Sprites und ihr Tempo in zwei Listen
fische = []
tempo = []
for i in range(4):
    fisch = pbc.sprite("fisch.png")
    fisch.set_to(random.randint(0, 200), random.randint(30, 200))
    fisch.show()
    fische.append(fisch)
    tempo.append(random.randint(1, 3))

hai = pbc.sprite("hai.png")
hai.set_to(60, 120)
hai.show()

krabbe = pbc.sprite("krabbe.png")
krabbe.set_to(20, 246)
krabbe.show()
krabbe_dx = 2

while True:
    for i in range(len(fische)):
        fische[i].set_by(tempo[i], 0)
        # Rechts hinaus? Dann links mit neuer Hoehe wieder herein.
        if fische[i].x > pbc.WIDTH:
            fische[i].set_to(-fische[i].width, random.randint(30, 200))
    hai.set_by(1, 0)
    if hai.x > pbc.WIDTH:
        hai.set_to(-hai.width, random.randint(40, 180))
    # Die Krabbe laeuft am Boden hin und her.
    krabbe.set_by(krabbe_dx, 0)
    if krabbe.x < 0 or krabbe.x + krabbe.width > 140:
        krabbe_dx = -krabbe_dx
    time.sleep_ms(30)
```



- Zeile 5 bis 7 machen das Meer-Bild zum Hintergrund.
- Zeile 10 bis 17 legen vier Fisch-Sprites an. Jeder bekommt eine zufällige Stelle und ein zufälliges Tempo. Sprites und Tempo stehen in zwei Listen: `tempo[i]` gehört zu `fische[i]`.
- In der Schleife schwimmt jeder Fisch um sein Tempo nach rechts. Ist er rechts hinaus (Zeile 32), setzt Zeile 33 ihn **links außerhalb** des Bildschirms wieder an, mit x = −Breite. Von dort schwimmt er wieder herein. Ein Sprite darf also teilweise außerhalb des Bildschirms stehen.
- Die Krabbe merkt sich ihre Richtung in `krabbe_dx`. Erreicht sie einen Rand ihres Wegs (Zeile 39), dreht Zeile 40 die Richtung um: Aus 2 wird −2.

> Jeder `set_by`-Befehl zeichnet den ganzen Bildschirm neu. Bewegen sich in jedem Schleifendurchlauf sechs Sprites, wird der Bildschirm sechsmal gezeichnet. Mit vielen Sprites läuft ein Programm deshalb langsamer.

Zum Verändern: Lass mehr Fische schwimmen. Gib dem Hai ein eigenes zufälliges Tempo. Lass die Krabbe auf dem ganzen Sandstreifen laufen.

Übungsaufgaben

Aufgabe 1 ⭐ Was passiert wann? Lies das Programm. Was siehst du in den ersten zwei Sekunden, was danach, und was steht im Terminal? Prüfe danach im Emulator.

k18_aufgabe1.py

```
1 import pbc
2 import time
3
4 stern = pbc.sprite("stern.png")
5 stern.set_to(100, 100)
6 time.sleep(1)
7 stern.glide_by(0, 100, 1000)
8 stern.show()
9 print(stern.x, stern.y)
```

Aufgabe 2 ⭐ Finde die Fehler Das Ufo soll von links nach rechts über den Bildschirm fliegen. Das Programm enthält drei Fehler. Nur einer erzeugt eine Fehlermeldung.

k18_aufgabe2.py

```
1 import pbc
2 import time
3
4 ufo = pbc.sprite("ufo")
5 ufo.set_to(0, 100)
6 for i in range(40):
7     ufo.x = ufo.x + 5
8     time.sleep_ms(30)
```

Aufgabe 3 ⭐⭐ Luftblasen Ergänze das Aquarium um drei Luftblasen (`kugel.png`). Sie steigen langsam nach oben. Ist eine oben hinaus, beginnt sie unten an einer zufälligen Stelle von neuem.

Aufgabe 4 ⭐⭐ Rundflug Das Ufo fliegt zweimal im Kreis um den Bildschirm: von Ecke zu Ecke und zurück zum Anfang. Speichere die Eckpunkte in einer Liste von Listen und benutze `glide_to` in einer Schleife. Nach jeder Runde steht im Terminal „Runde 1 fertig“ bzw. „Runde 2 fertig“.

Aufgabe 5 ⭐⭐⭐ Rakete mit Tank Steuere die Rakete mit dem Joystick über den Weltraum-Hintergrund. Solange Taste A gedrückt ist, fliegt sie dreimal so schnell (Turbo). Jede Bewegung verbraucht Treibstoff, Turbo mehr als normal. Links oben zeigt ein grüner Balken im Hintergrund-Canvas, wie voll der Tank ist. Ist er leer, erscheint „Tank leer!“ und das Programm endet.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Bilder für Sprites lädst du über **Dateien → Bilder verwalten… → PNG hochladen…** hoch, für den Emulator und das Gerät getrennt.
- `s = pbc.sprite("bild.png")` legt ein Sprite an. Es steht bei (0, 0) und ist unsichtbar, bis `s.show()` aufgerufen wird. `s.hide()` versteckt es.
- Bewegen: `set_to(x, y)`, `set_by(dx, dy)`, `glide_to(x, y, ms)`, `glide_by(dx, dy, ms)`. Gleiten wartet, bis es fertig ist.
- `s.x`, `s.y`, `s.width`, `s.height` lesen; nicht direkt zuweisen.
- Sprites werden automatisch neu gezeichnet: Hintergrund, dann alle sichtbaren Sprites.
- Hintergrund: `pbc.Sprite._background = canvas` und `pbc.show(canvas)`. Texte gehören in den Hintergrund-Canvas, sonst werden sie übermalt.

KAPITEL 19

Kollisionen

Worum geht's? In fast jedem Spiel stoßen Dinge zusammen: Die Spielfigur sammelt eine Münze ein, ein Ball prallt vom Rand ab, wer in die Lava tritt, muss zurück zum Start. Solche Zusammenstöße heißen **Kollisionen**. Sprites können drei Arten von Kollisionen selbst erkennen. Außerdem lernst du, wie du Kollisionen mit eigenen Funktionen ausrechnest.

Für dieses Kapitel brauchst du die Bilder `held.png`, `muenze.png`, `kugel.png`, `ufo.png`, `stern.png`, `schlaeger.png` und `pingpong.png` (hochladen wie in Abschnitt 18.1).

19.1 Sprite berührt Sprite: `touches`

`a.touches(b)` ist `True`, wenn das Sprite `a` das Sprite `b` berührt, sonst `False`. Damit sammelt der Held hier Münzen ein:

k19_muenzen.py

```
import pbc
import random
import time

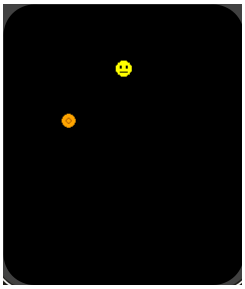
held = pbc.sprite("held.png")
held.set_to(112, 200)
held.show()

muenze = pbc.sprite("muenze.png")
muenze.set_to(113, 60)
muenze.show()

punkte = 0
while True:
    if pbc.pressed_left():
        held.set_by(-3, 0)
    if pbc.pressed_right():
        held.set_by(3, 0)
    if pbc.pressed_up():
        held.set_by(0, -3)
    if pbc.pressed_down():
        held.set_by(0, 3)
    if held.touches(muenze):
        punkte = punkte + 1
        print("Punkte:", punkte)
        pbc.tone(1200, 80)
        muenze.set_to(random.randint(10, 210), random.randint(10, 250))
    time.sleep_ms(20)
```

Terminal

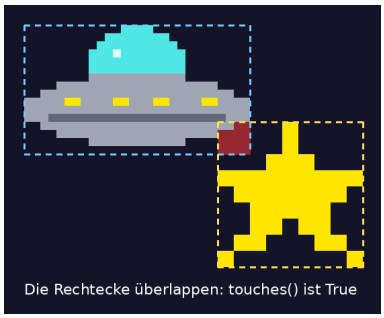
```
Punkte: 1
```



- Zeile 15 bis 22 bewegen den Helden mit dem Joystick.
- Zeile 23 fragt in jedem Durchlauf der Spielschleife: Berührt der Held die Münze?
- Wenn ja, gibt es einen Punkt und einen kurzen Ton, und Zeile 27 setzt die Münze an eine zufällige neue Stelle. Weil sie dann nicht mehr berührt wird, zählt jede Münze nur einmal.

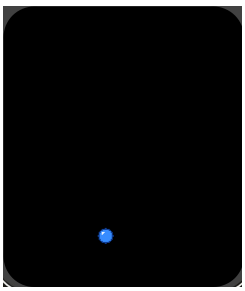
Wann „berühren" sich zwei Sprites? `touches` vergleicht die **Rechtecke** der beiden Bilder, nicht ihre Formen. Auch die durchsichtigen Ecken eines Bildes gehören zum Rechteck. Überlappen sich die Rechtecke, ist das Ergebnis `True`, selbst wenn sich die sichtbaren Figuren noch gar nicht berühren:

Für die meisten Spiele ist das genau genug. Bei Figuren mit großen durchsichtigen Ecken wirkt es manchmal, als würde schon ein „Beinahe-Treffer" zählen. Dann hilft es, das Bild enger zuzuschneiden.



19.2 Sprite berührt Rand: `touches_edge`

`s.touches_edge()` ist `True`, wenn ein Teil des Sprites **außerhalb** des Bildschirms liegt. Ein Sprite, das genau am Rand anliegt, zählt noch nicht. Damit lässt du eine Kugel an den Rändern abprallen:



k19_rand.py

```
 1 import pbc
 2 import time
 3
 4 kugel = pbc.sprite("kugel.png")
 5 kugel.set_to(50, 80)
 6 kugel.show()
 7 dx = 4
 8 dy = 3
 9
10 while True:
11     # Waagrecht bewegen; am Rand zurueck und umdrehen
12     kugel.set_by(dx, 0)
13     if kugel.touches_edge():
14         kugel.set_by(-dx, 0)
15         dx = -dx
16     # Senkrecht bewegen; am Rand zurueck und umdrehen
17     kugel.set_by(0, dy)
18     if kugel.touches_edge():
19         kugel.set_by(0, -dy)
20         dy = -dy
21     time.sleep_ms(10)
```

Die Kugel merkt sich ihre Richtung in `dx` und `dy`. Der Trick: Sie bewegt sich **erst waagrecht, dann senkrecht**, und prüft nach jedem Teilschritt den Rand.

- Zeile 12 bewegt waagrecht. Ragt die Kugel jetzt hinaus, war es der linke oder rechte Rand. Zeile 14 nimmt den Schritt zurück, Zeile 15 dreht die waagrechte Richtung um.
- Zeile 17 bis 20 machen dasselbe senkrecht für oben und unten.
- So weiß das Programm immer, welche Richtung es umdrehen muss, ohne die Koordinaten selbst zu vergleichen.

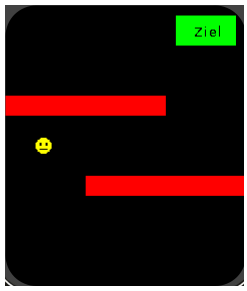
`touches_edge` sagt also nur, **dass** das Sprite hinausragt, nicht **wo**. Willst du wissen, an welchem Rand, vergleichst du wie in Kapitel 18 selbst: `s.x < 0` (links), `s.x + s.width > pbc.WIDTH` (rechts), `s.y < 0` (oben), `s.y + s.height > pbc.HEIGHT` (unten).

19.3 Sprite berührt Farbe: `on_color`

`s.on_color(farbe)` ist `True`, wenn das Sprite über einer Stelle des **Hintergrund-Canvas** steht, die genau diese Farbe hat. Damit baust du Spielfelder mit gefährlichen oder besonderen Bereichen, ohne ihre Lage im Programm ausrechnen zu müssen:

k19_lava.py

```
 1 import pbc
 2 import time
 3
 4 # Spielfeld: schwarzer Boden, rote Lava, gruenes Ziel
 5 feld = pbc.Canvas(240, 280)
 6 feld.fill(pbc.BLACK)
 7 feld.fill_rect(0, 90, 160, 20, pbc.RED)
 8 feld.fill_rect(80, 170, 160, 20, pbc.RED)
 9 feld.fill_rect(170, 10, 60, 30, pbc.GREEN)
10 feld.text12("Ziel", 188, 18, pbc.BLACK)
11 pbc.Sprite._background = feld
12 pbc.show(feld)
13
14 held = pbc.sprite("held.png")
15 held.set_to(30, 240)
16 held.show()
17
18 while True:
19     if pbc.pressed_left():
20         held.set_by(-2, 0)
21     if pbc.pressed_right():
22         held.set_by(2, 0)
23     if pbc.pressed_up():
24         held.set_by(0, -2)
25     if pbc.pressed_down():
26         held.set_by(0, 2)
27     if held.on_color(pbc.RED):
28         print("Autsch! Zurueck zum Start.")
29         held.set_to(30, 240)
30     if held.on_color(pbc.GREEN):
31         print("Geschafft!")
32         break
```



```
    time.sleep_ms(20)

feld.text24("Geschafft!", 62, 130, pbc.YELLOW)
pbc.show(feld)
```

Terminal

```
Autsch! Zurueck zum Start.
Autsch! Zurueck zum Start.
```

- Zeile 5 bis 12 zeichnen das Spielfeld in einen Canvas und machen ihn zum Hintergrund der Sprites (Abschnitt 18.5).
- Zeile 27: Steht der Held auf Rot, muss er zurück zum Start.
- Zeile 30: Erreicht er das grüne Ziel, verlässt `break` die Schleife, und „Geschafft!“ erscheint im Hintergrund.

So genau prüft `on_color`:

- Es zählen nur die **sichtbaren** Pixel des Sprites. Die durchsichtigen Ecken dürfen über der Lava stehen. Anders als bei `touches` kommt es hier also auf die Form an.
- Geprüft wird nur der **Hintergrund-Canvas**, nicht der Bildschirm und nicht die anderen Sprites. Ohne `pbc.Sprite._background` ist das Ergebnis immer `False`.
- Die Farbe muss **genau** stimmen. Zeichne Bereiche, auf die du prüfst, deshalb mit den Farbkonstanten (`pbc.RED`, `pbc.GREEN` …). Bei einem geladenen Bild als Hintergrund müsste die Farbe im PNG ebenfalls genau stimmen (zum Beispiel 255, 0, 0 für `pbc.RED`) und darf an den Rändern nicht geglättet sein.

> Zeichnest du die Lava mit `pbc.fill_rect` direkt auf den Bildschirm, passiert zweierlei: Sobald sich ein Sprite bewegt, ist die Lava weg (Abschnitt 18.5), und `on_color` findet sie nie, weil sie nicht im Hintergrund-Canvas steht.

19.4 Kollisionen selbst berechnen: Abstand und Rechtecke

Arbeitest du ohne Sprites, zum Beispiel mit Kreisen auf einem Canvas, rechnest du Kollisionen mit eigenen Funktionen aus (Kapitel 12).

Der Abstand zweier Punkte. Liegen zwei Punkte `dx` Pixel waagrecht und `dy` Pixel senkrecht auseinander, ist ihr Abstand nach dem Satz des Pythagoras $\sqrt{(dx^2 + dy^2)}$. Für die Wurzel gibt es das Modul `math` mit der Funktion `math.sqrt` (englisch *square root*). Wie `pbc` musst du es oben im Programm importieren.

Zwei Kreise berühren sich, wenn der Abstand ihrer Mittelpunkte höchstens so groß ist wie die Summe ihrer Radien. Statt die Wurzel zu ziehen, kannst du auch beide Seiten **quadrieren** und `dx * dx + dy * dy` mit `(r1 + r2) * (r1 + r2)` vergleichen. Das Ergebnis ist dasselbe, und der PicoBoy muss keine Wurzel ausrechnen.

Zwei Rechtecke überlappen sich, wenn sie sich waagrecht **und** senkrecht überlappen. Waagrecht überlappen sie, wenn jedes Rechteck links vor dem rechten Rand des anderen beginnt. Genauso rechnet `touches`.

k19_abstand.py

```python
import math


def abstand(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    return math.sqrt(dx * dx + dy * dy)


def kreise_beruehren(x1, y1, r1, x2, y2, r2):
    # Vergleich im Quadrat: kein math.sqrt noetig
    dx = x2 - x1
    dy = y2 - y1
    return dx * dx + dy * dy <= (r1 + r2) * (r1 + r2)


def rechtecke_ueberlappen(x1, y1, b1, h1, x2, y2, b2, h2):
    return (x1 < x2 + b2 and x2 < x1 + b1 and
            y1 < y2 + h2 and y2 < y1 + h1)


print(abstand(0, 0, 3, 4))
print(abstand(10, 20, 10, 50))
print(kreise_beruehren(0, 0, 10, 15, 0, 5))
print(kreise_beruehren(0, 0, 10, 15, 15, 5))
print(rechtecke_ueberlappen(0, 0, 28, 16, 24, 12, 18, 18))
print(rechtecke_ueberlappen(0, 0, 28, 16, 28, 12, 18, 18))
```

Terminal

```
5.0
30.0
True
False
True
False
```

- `abstand(0, 0, 3, 4)` ist 5,0: das bekannte 3-4-5-Dreieck. `math.sqrt` liefert immer eine Kommazahl.
- Die Kreise mit den Radien 10 und 5 berühren sich bei einem Abstand von 15 gerade noch (`<=`). Bei (15, 15) ist der Abstand etwa 21,2, also zu groß.
- Die Rechteck-Beispiele sind Ufo und Stern aus der Abbildung in 19.1. Rückt der Stern nach x = 28, beginnt er genau dort, wo das Ufo aufhört (0 + 28). Dann überlappen sie nicht mehr, deshalb steht im Vergleich `<` und nicht `<=`.

Die Ergebnisse von Hand nachzuprüfen, bevor man eine Funktion in einem Spiel benutzt, ist eine gute Gewohnheit (Abschnitt 12.5). In Übungsaufgabe 4 schreibst du selbst eine solche Funktion.

Ausprobieren: Ball und Schläger

Ein Ball prallt von den Wänden und vom Schläger ab. Mit dem Joystick bewegst du den Schläger. Fällt der Ball unten hinaus, ist das Spiel vorbei.

k19_pingpong.py

```
import pbc
import random
import time

feld = pbc.Canvas(240, 280)
pbc.Sprite._background = feld

schlaeger = pbc.sprite("schlaeger.png")
schlaeger.set_to(98, 260)
schlaeger.show()

ball = pbc.sprite("pingpong.png")
ball.set_to(random.randint(20, 200), 40)
ball.show()

dx = 3
dy = 3
punkte = 0


def zeige_punkte():
    feld.fill(pbc.rgb(0, 70, 40))
    feld.text18(f"Punkte: {punkte}", 10, 8, pbc.WHITE)
    pbc.show(feld)


zeige_punkte()
while True:
    # Schlaeger steuern, ohne den Bildschirm zu verlassen
    if pbc.pressed_left() and schlaeger.x >= 6:
        schlaeger.set_by(-6, 0)
    if pbc.pressed_right() and schlaeger.x + schlaeger.width + 6 <= pbc.WIDTH:
        schlaeger.set_by(6, 0)

    # Ball waagrecht: Abprallen am linken und rechten Rand
    ball.set_by(dx, 0)
    if ball.touches_edge():
        ball.set_by(-dx, 0)
        dx = -dx
    # Ball senkrecht: unten hinaus ist verloren, oben prallt er ab
    ball.set_by(0, dy)
    if ball.y + ball.height > pbc.HEIGHT:
        break
    if ball.touches_edge():
        ball.set_by(0, -dy)
        dy = -dy
```

```
48      # Ball trifft den Schlaeger (nur auf dem Weg nach unten)
49      if ball.touches(schlaeger) and dy > 0:
50          dy = -dy
51          punkte = punkte + 1
52          pbc.tone(880, 50)
53          zeige_punkte()
54      time.sleep_ms(10)
55
56  feld.text24("Game over", 65, 120, pbc.YELLOW)
57  pbc.show(feld)
58  print("Punkte:", punkte)
```



- Der Hintergrund ist ein eigener Canvas (Zeile 5 und 6). `zeige_punkte` füllt ihn neu, schreibt den Punktestand hinein und zeigt ihn.
- Zeile 30 bis 33 bewegen den Schläger und halten ihn im Bildschirm.
- Zeile 36 bis 46 bewegen den Ball wie in Abschnitt 19.2. Neu ist Zeile 42: Ist der Ball **unten** hinaus, verlässt `break` die Schleife, **bevor** `touches_edge` ihn abprallen lassen könnte.
- Zeile 49: Trifft der Ball den Schläger, fliegt er wieder nach oben. Die Bedingung `dy > 0` sorgt dafür, dass nur ein Ball zählt, der gerade nach unten fliegt. Sonst könnte ein Ball, der noch im Schläger steckt, mehrmals hintereinander umgedreht werden.

Zum Verändern: Mach den Ball nach jeweils fünf Punkten schneller. Lass den Ball je nach Trefferstelle auf dem Schläger schräger abprallen: Trifft er die linke Hälfte, fliegt er nach links.

Übungsaufgaben

Aufgabe 1 ⭐ Berührt oder nicht? Das Ufo ist 28 × 16 Pixel groß, der Stern 18 × 18. Was gibt das Programm aus? Rechne mit den Rechtecken und prüfe danach im Emulator.

k19_aufgabe1.py

```
 1  import pbc
 2
 3  ufo = pbc.sprite("ufo.png")      # 28 x 16 Pixel
 4  stern = pbc.sprite("stern.png")  # 18 x 18 Pixel
 5
 6  ufo.set_to(0, 0)
 7  stern.set_to(24, 12)
 8  print(ufo.touches(stern))
 9  stern.set_to(28, 12)
10  print(ufo.touches(stern))
11  stern.set_to(10, 15)
12  print(stern.touches(ufo))
13  print(ufo.touches_edge(), stern.touches_edge())
14  ufo.set_to(212, 100)
15  print(ufo.touches_edge())
```

Aufgabe 2 ⭐ Finde den Fehler Die Lava verschwindet, sobald der Held losläuft, und „Lava!“ erscheint nie. Was ist falsch? Verbessere das Programm.

k19_aufgabe2.py

```
import pbc
import time

pbc.fill_rect(0, 120, 240, 40, pbc.RED)
held = pbc.sprite("held.png")
held.set_to(112, 250)
held.show()

while True:
    if pbc.pressed_up():
        held.set_by(0, -2)
    if held.on_color(pbc.RED):
        print("Lava!")
        held.set_to(112, 250)
    time.sleep_ms(20)
```

Aufgabe 3 ⭐⭐ Drei Leben Ändere „Ball und Schläger“: Fällt der Ball unten hinaus, kostet das ein Leben, und der Ball startet oben neu. Nach drei verlorenen Leben ist Schluss. Zeige die Leben neben den Punkten an.

Aufgabe 4 ⭐⭐ Punkt im Rechteck Schreibe `punkt_im_rechteck(px, py, x, y, b, h)`: `True`, wenn (px, py) im Rechteck mit linker oberer Ecke (x, y), Breite b und Höhe h liegt. Teste mit fünf Punkten, auch genau auf dem Rand und gleich daneben.

Aufgabe 5 ⭐⭐⭐ Meteoritenregen Ein Raumschiff (`raumschiff.png`) weicht unten vor dem Weltraum nach links und rechts aus. Drei Meteore (`meteor.png`) fallen herab und starten oben zufällig neu. Ein Treffer beendet das Spiel; dann siehst du, wie viele Sekunden du durchgehalten hast (`time.ticks_ms`).

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `a.touches(b)`: Überlappen sich die Rechtecke der Bilder?
- `s.touches_edge()`: Ragt das Sprite über den Bildschirmrand hinaus?
- `s.on_color(farbe)`: Steht ein sichtbares Pixel des Sprites über genau dieser Farbe im Hintergrund-Canvas?
- Selbst rechnen: Abstand mit `math.sqrt(dx * dx + dy * dy)`; Kreise berühren sich, wenn `dx * dx + dy * dy <= (r1 + r2) * (r1 + r2)`.

KAPITEL 20

Der Bewegungssensor

Worum geht's? Im PicoBoy steckt ein Sensor, der merkt, wie das Gerät gehalten, gekippt oder geschüttelt wird. Du liest seine Werte aus, steuerst eine Kugel durch Kippen, glättest zittrige Messwerte und baust eine Wasserwaage.

> **Diese Programme brauchen das echte Gerät.** Im Emulator liegt der PicoBoy immer flach auf dem Tisch. Der Sensor liefert dort immer dieselben Werte: x = 0, y = 0, z = 9,81. Alle Programme dieses Kapitels laufen im Emulator, aber es bewegt sich nichts. Teste sie deshalb auf dem PicoBoy (Ziel **PBC+**, **Verbinden**, ▶ **Testen**). Die Bilder in diesem Kapitel stammen aus dem Emulator und zeigen deshalb immer die flache Lage.

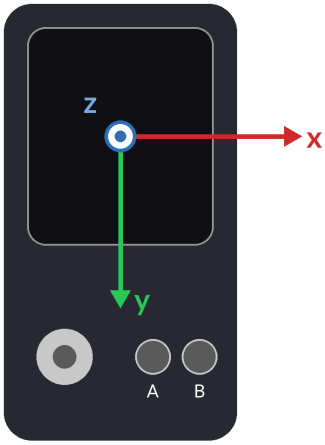
20.1 Was misst der Sensor?

Der Bewegungssensor ist ein **Beschleunigungssensor**. Er misst, wie stark das Gerät in eine Richtung beschleunigt wird. Die Einheit ist **Meter pro Sekunde zum Quadrat** (m/s^2). Das Besondere: Auch die **Erdanziehung** wirkt auf den Sensor wie eine Beschleunigung. Liegt der PicoBoy ruhig auf dem Tisch, misst der Sensor deshalb etwa 9,81 m/s^2. So stark zieht die Erde an allem, was auf ihr liegt. Dieser Wert heißt auch **1 g** (g für *gravity*, Schwerkraft).

Der Sensor misst in drei Richtungen, den **Achsen**:

- **x** zeigt nach **rechts**,
- **y** zeigt zum **unteren** Rand des Bildschirms (wie die y-Koordinate auf dem Display),
- **z** zeigt aus dem Bildschirm **heraus**, auf dich zu.

So steht es in der Firmware des PicoBoy. Für jede Achse liefert der Sensor eine Zahl. Zeigt eine Achse nach oben, misst sie die Erdanziehung als **positiven** Wert, zeigt sie nach unten, als **negativen**. Zeigt sie zur Seite, misst sie 0. Daraus ergeben sich diese Richtwerte für ein ruhig gehaltenes Gerät:



z zeigt aus dem Bildschirm heraus (auf dich zu)

Lage des PicoBoy	x	y	z
flach, Bildschirm nach oben	0	0	+9,8
flach, Bildschirm nach unten	0	0	−9,8
aufrecht wie ein Bild an der Wand	0	−9,8	0
auf der linken Kante, rechter Rand oben	+9,8	0	0

Echte Messwerte weichen davon etwas ab und schwanken ein wenig. Liegt das Gerät schräg, verteilt sich die Erdanziehung auf mehrere Achsen.

Bewegst du den PicoBoy, kommt zur Erdanziehung die Beschleunigung der Bewegung dazu. Beim Schütteln springen die Werte deshalb stark hin und her. Laut Firmware misst der Sensor bis etwa 2 g, also ungefähr 19,6 m/s² in jede Richtung; stärkere Stöße werden abgeschnitten.

20.2 `accel_x`, `accel_y`, `accel_z`

Drei Funktionen im Modul `pbc` lesen den Sensor aus. Jede liefert eine Kommazahl in m/s². Dieselben Werte in g liefern `accel_x_g()`, `accel_y_g()` und `accel_z_g()`: Dort bedeutet 1.0 genau die Erdanziehung.

k20_werte.py

```
 1 import pbc
 2
 3 print("In m/s²:")
 4 print("x =", pbc.accel_x())
 5 print("y =", pbc.accel_y())
 6 print("z =", pbc.accel_z())
 7 print("In g:")
 8 print("x =", pbc.accel_x_g())
 9 print("y =", pbc.accel_y_g())
10 print("z =", pbc.accel_z_g())
```

Terminal (Emulator)

```
In m/s²:
x = 0.0
y = 0.0
z = 9.81
In g:
x = 0.0
y = 0.0
z = 1.0
```

Im Emulator sind die Werte immer gleich: Das Gerät liegt flach, Bildschirm nach oben. Auf dem echten PicoBoy bekommst du bei jedem Aufruf einen neuen Messwert.

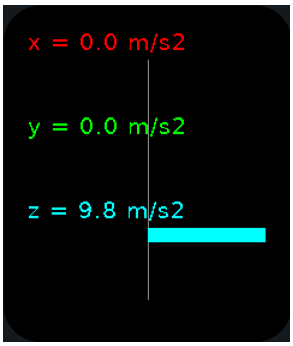
Damit du die Werte live siehst, zeigt das nächste Programm sie fortlaufend als Zahl und als Balken. Es zeichnet auf einem Canvas (Kapitel 17), damit nichts flackert:

k20_anzeige.py

```
 1 import pbc
 2 import time
 3
 4 bild = pbc.Canvas(240, 280)
 5 MITTE = 120
 6
 7 def balken(name, wert, y, farbe):
 8     bild.text18(f"{name} = {wert:.1f} m/s2", 20, y, farbe)
 9     laenge = int(wert * 10)     # 1 m/s² = 10 Pixel
```

```
    if laenge >= 0:
        bild.fill_rect(MITTE, y + 25, laenge, 12, farbe)
    else:
        bild.fill_rect(MITTE + laenge, y + 25, -laenge, 12, farbe)

while True:
    bild.fill(pbc.BLACK)
    bild.vline(MITTE, 45, 200, pbc.GRAY)
    balken("x", pbc.accel_x(), 20, pbc.RED)
    balken("y", pbc.accel_y(), 90, pbc.GREEN)
    balken("z", pbc.accel_z(), 160, pbc.CYAN)
    pbc.show(bild)
    time.sleep_ms(50)
```



- Zeile 8: `{wert:.1f}` im f-String zeigt die Zahl mit **einer** Stelle nach dem Komma. Mit `.2f` wären es zwei Stellen.
- Zeile 9–13: 1 m/s² wird 10 Pixel lang. Positive Werte zeichnen einen Balken nach rechts von der grauen Mittellinie, negative nach links. Weil ein Rechteck keine negative Breite haben soll, beginnt der linke Balken bei `MITTE + laenge` und ist `-laenge` breit.
- Zeile 15–22: Die Schleife misst und zeichnet etwa 20-mal pro Sekunde.

Auf dem Gerät ausprobieren: Kippe den PicoBoy langsam nach links und rechts, stelle ihn hin, dreh ihn um. Welche Achse ändert sich? Stimmen die Vorzeichen mit der Tabelle in 20.1 überein? Notiere, welche Werte dein Gerät ungefähr zeigt, wenn es ruhig auf dem Tisch liegt.

20.3 Kippen statt drücken

Mit dem Sensor kannst du ein Spiel durch **Kippen** steuern. Die einfachste Art: Der Messwert wird wie eine Taste behandelt. Ist er stärker als eine **Grenze**, gilt das als „gedrückt“.

k20_kipprichtung.py

```
import pbc
import time

GRENZE = 3      # m/s²: so stark muss man kippen
alt = ""

while True:
    x = pbc.accel_x()
    y = pbc.accel_y()
    # die tiefere Seite misst einen negativen Wert
    if x < -GRENZE:
        richtung = "rechts"
    elif x > GRENZE:
        richtung = "links"
    elif y < -GRENZE:
        richtung = "unten"
```



```
    elif y > GRENZE:
        richtung = "oben"
    else:
        richtung = "flach"
    # nur bei einer Änderung neu zeichnen (kein Flackern)
    if richtung != alt:
        pbc.clear()
        pbc.text24(richtung, 85, 125, pbc.YELLOW)
        alt = richtung
    time.sleep_ms(100)
```

- Zeile 4: Unterhalb von 3 m/s² gilt das Gerät als flach. So reagiert das Programm nicht auf jedes kleine Zittern.
- Zeile 11–20: Senkst du den rechten Rand, zeigt die x-Achse schräg nach unten, und x wird negativ (siehe 20.1). Darum steht bei `x < -GRENZE` das Wort „rechts“.
- Zeile 22–25: Der Bildschirm wird nur neu gezeichnet, wenn sich die Richtung geändert hat. Das verhindert Flackern.

> Reagiert dein Programm auf dem Gerät „falsch herum“, tausche einfach `<` und `>` oder die Wörter. Prüfe die Richtung immer mit dem Programm aus 20.2.

Je schräger, desto schneller. Statt nur „gekippt oder nicht“ kann der Messwert auch direkt die Geschwindigkeit bestimmen. Diese Kugel rollt wie auf einem Tablett:

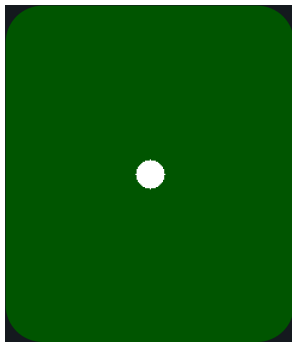
k20_kugel.py

```
import pbc
import time

bild = pbc.Canvas(240, 280)
RADIUS = 12
TEMPO = 0.8     # Pixel pro m/s² und Durchlauf
x = 120
y = 140

while True:
    # Die Kugel rollt zur tiefen Seite, also gegen den Messwert
    x = x - pbc.accel_x() * TEMPO
    y = y - pbc.accel_y() * TEMPO
    # am Rand anhalten
    x = max(RADIUS, min(239 - RADIUS, x))
    y = max(RADIUS, min(279 - RADIUS, y))

    bild.fill(pbc.rgb(0, 90, 0))
    bild.ellipse(int(x), int(y), RADIUS, RADIUS, pbc.WHITE, True)
    pbc.show(bild)
    time.sleep_ms(30)
```



- Zeile 12–13: Die Kugel rollt zur **tieferen** Seite. Ist der rechte Rand tiefer, ist x negativ. Damit die Kugel trotzdem nach rechts rollt, ziehen wir den Messwert ab: `x - (negativ)` wird größer. `TEMPO` legt fest, wie viele Pixel die Kugel pro m/s² und Durchlauf rollt.
- Zeile 15–16: `min(239 - RADIUS, x)` liefert den kleineren der beiden Werte, `max(RADIUS, …)` den größeren. Zusammen halten sie die Kugel auf dem Bildschirm.
- Zeile 19: Die Position ist eine Kommazahl. Zum Zeichnen machen wir mit `int()` eine ganze Zahl daraus.

20.4 Werte glätten

Die Messwerte eines echten Sensors zittern: Selbst wenn das Gerät ruhig liegt, ändern sie sich von Messung zu Messung ein wenig. Eine Anzeige oder eine Spielfigur zittert dann mit. Abhilfe schafft **Glätten**: Man nimmt nicht jeden neuen Wert ganz, sondern nur einen Teil davon.

```
glatt = glatt + (roh - glatt) * 0.2
```

`roh` ist der neue Messwert, `glatt` der geglättete Wert. `roh - glatt` ist der Unterschied zwischen beiden. Davon kommen nur 20 % dazu. Ein einzelner Ausreißer verschiebt `glatt` deshalb nur wenig. Bleibt der neue Wert aber länger bestehen, wandert `glatt` Schritt für Schritt dorthin.

Im Emulator zittert nichts. Damit du die Wirkung trotzdem siehst, mischt dieses Programm ein künstliches Zittern zum Messwert:

k20_glaetten.py

```
 1 import pbc
 2 import random
 3
 4 ANTEIL = 0.2        # wie stark jeder neue Wert zählt
 5 glatt = pbc.accel_x()
 6
 7 for i in range(12):
 8     # Im Emulator ist x immer 0. Damit man etwas sieht, mischen wir
 9     # ein künstliches Zittern dazu (auf dem Gerät weglassen).
10     roh = pbc.accel_x() + random.randint(-20, 20) / 10
11     glatt = glatt + (roh - glatt) * ANTEIL
12     print(f"roh: {roh:5.1f}   geglaettet: {glatt:5.2f}")
```

Terminal (Emulator)

```
roh:  -1.1   geglaettet: -0.22
roh:   1.8   geglaettet:  0.18
roh:   0.4   geglaettet:  0.23
roh:   0.6   geglaettet:  0.30
roh:   0.9   geglaettet:  0.42
roh:  -1.1   geglaettet:  0.12
roh:  -2.0   geglaettet: -0.31
roh:   1.3   geglaettet:  0.01
```

```
roh:  -1.1   geglaettet: -0.21
roh:  -0.4   geglaettet: -0.25
roh:   0.1   geglaettet: -0.18
roh:  -1.9   geglaettet: -0.52
```

- Zeile 10: `random.randint(-20, 20) / 10` ist eine zufällige Zahl zwischen −2,0 und 2,0. Auf dem Gerät lässt du sie weg.
- Zeile 11: die Glättung. Die rohen Werte springen um bis zu 2 hin und her, die geglätteten bleiben viel näher bei 0.
- `{roh:5.1f}`: Die 5 sorgt dafür, dass jede Zahl mindestens 5 Zeichen breit ist. So stehen die Spalten untereinander.

Wie stark glätten? Ein kleiner Anteil (z. B. 0,1) glättet stark, aber der Wert folgt einer echten Bewegung nur langsam. Ein großer Anteil (z. B. 0,5) reagiert schnell, zittert aber mehr. Probiere auf dem Gerät aus, was zu deinem Programm passt.

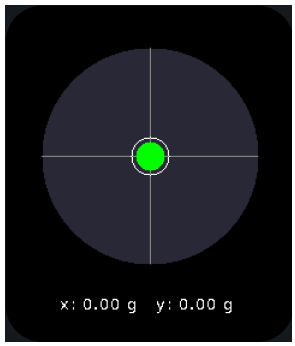
Ausprobieren: Die Wasserwaage

Eine Wasserwaage hat eine kleine Luftblase in einem Röhrchen mit Flüssigkeit. Die Blase steigt immer zur **höchsten** Stelle. Unser PicoBoy macht daraus eine runde Wasserwaage: Liegt er genau waagrecht, steht die Blase in der Mitte und wird grün.

k20_wasserwaage.py

```
 1 import pbc
 2 import time
 3
 4 bild = pbc.Canvas(240, 280)
 5 MX = 120            # Mitte der Libelle
 6 MY = 125
 7 WEG = 75            # so weit darf die Blase wandern
 8 glatt_x = 0
 9 glatt_y = 0
10
11 while True:
12     # Messwerte in g, geglättet
13     glatt_x = glatt_x + (pbc.accel_x_g() - glatt_x) * 0.3
14     glatt_y = glatt_y + (pbc.accel_y_g() - glatt_y) * 0.3
15
16     # Die Blase steigt zur hohen Seite: gleiche Richtung wie der Messwert
17     bx = MX + int(glatt_x * 200)
18     by = MY + int(glatt_y * 200)
19     bx = max(MX - WEG, min(MX + WEG, bx))
20     by = max(MY - WEG, min(MY + WEG, by))
21
22     waagrecht = abs(glatt_x) < 0.03 and abs(glatt_y) < 0.03
23     farbe = pbc.YELLOW
24     if waagrecht:
25         farbe = pbc.GREEN
```

- Zeile 13–14: Die Messwerte in g werden geglättet (Abschnitt 20.4).
- Zeile 17–18: Hebst du den rechten Rand an, wird x positiv, und die Blase wandert nach rechts, also zur hohen Seite. Anders als die Kugel aus 20.3 folgt die Blase deshalb dem Messwert mit **gleichem** Vorzeichen. 0,1 g Neigung verschiebt die Blase um 20 Pixel.
- Zeile 19–20: Die Blase bleibt im großen Kreis.
- Zeile 22–25: Weniger als 0,03 g auf beiden Achsen gilt als waagrecht. Dann wird die Blase grün.



k20_wasserwaage.py

```
27      bild.fill(pbc.BLACK)
28      bild.ellipse(MX, MY, WEG + 15, WEG + 15, pbc.rgb(40, 40, 60), True)
29      bild.hline(MX - 90, MY, 180, pbc.GRAY)
30      bild.vline(MX, MY - 90, 180, pbc.GRAY)
31      bild.ellipse(MX, MY, 16, 16, pbc.WHITE)
32      bild.ellipse(bx, by, 12, 12, farbe, True)
33      bild.text12(f"x: {glatt_x:.2f} g   y: {glatt_y:.2f} g", 45, 240, pbc.WHITE)
34      pbc.show(bild)
35      pbc.led_green(waagrecht)
36      time.sleep_ms(40)
```

Hintergrund, Kreis, Fadenkreuz, Zielkreis und Blase werden auf dem Canvas gezeichnet und mit `pbc.show` auf einmal angezeigt. Zeile 35 schaltet die grüne LED ein, solange die Waage waagrecht liegt.

Im Emulator steht die Blase immer grün in der Mitte. Auf dem Gerät kannst du damit prüfen, ob dein Tisch gerade ist.

Zum Verändern: Mach die Wasserwaage empfindlicher (größerer Faktor als 200) oder strenger (kleinere Grenze als 0,03). Lass zusätzlich einen kurzen Ton erklingen, sobald die Waage waagrecht ist.

Übungsaufgaben

Aufgabe 1 ⭐ Wie liegt das Gerät? Was gibt das Programm im Emulator aus? Was gibt es auf dem Gerät aus, wenn der PicoBoy mit dem Bildschirm nach unten auf dem Tisch liegt, und was, wenn er aufrecht steht?

k20_aufgabe1.py

```
1 import pbc
2
3 z = pbc.accel_z()
4 if z > 8:
5     print("Bildschirm zeigt nach oben")
6 elif z < -8:
7     print("Bildschirm zeigt nach unten")
8 else:
9     print("Das Gerät steht oder ist stark gekippt")
```

Aufgabe 2 ⭐ Kipp-Ampel Schreibe ein Programm, das die RGB-LED grün leuchten lässt, solange der PicoBoy waagrecht liegt. Ist der linke Rand deutlich tiefer, leuchtet sie blau, ist der rechte Rand tiefer, rot. Teste auf dem Gerät.

Aufgabe 3 ⭐⭐ Schüttelwürfel Wenn man den PicoBoy schüttelt, zeigt er eine zufällige Augenzahl von 1 bis 6. Tipp: Merke dir die Werte der letzten Messung. Ändert sich die Summe der drei Unterschiede (mit `abs`) um mehr als 8 m/s², wurde geschüttelt. Damit du auch im Emulator testen kannst, soll zusätzlich Taste A würfeln.

Aufgabe 4 ⭐⭐ Mittelwert der letzten fünf Eine andere Art zu glätten: Speichere die letzten fünf Messwerte in einer Liste und nimm ihren Mittelwert. Kommt ein neuer Wert dazu, fällt der älteste heraus. Nimm `k20_glaetten.py` als Vorlage (mit künstlichem Zittern).

Aufgabe 5 ⭐⭐⭐ Münzen einsammeln Erweitere die rollende Kugel aus 20.3 zu einem Spiel: An einer zufälligen Stelle liegt eine Münze. Rollt die Kugel darüber, gibt es einen Punkt, einen Ton, und die Münze springt an eine neue Stelle. Prüfe die Berührung mit dem Abstand der Mittelpunkte (Kapitel 19). Damit du im Emulator testen kannst, soll die Kugel zusätzlich mit dem Joystick fahren.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

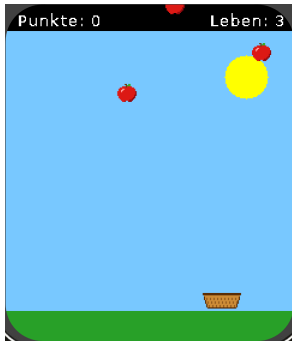
- Der Bewegungssensor misst Beschleunigungen in drei Richtungen, auch die Erdanziehung: In Ruhe zeigt die Achse, die nach oben zeigt, etwa +9,81 m/s² (= 1 g).
- Achsen: x nach rechts, y zum unteren Bildschirmrand, z aus dem Bildschirm heraus.
- `pbc.accel_x()`, `accel_y()`, `accel_z()` liefern m/s², `pbc.accel_x_g()` … dasselbe in g.
- Kippen statt drücken: Messwert mit einer Grenze vergleichen, oder den Messwert direkt als Geschwindigkeit nehmen.
- Glätten: `glatt = glatt + (roh - glatt) * anteil`.
- Im Emulator liegt das Gerät immer flach (0, 0, 9,81). Programme mit dem Sensor auf dem echten PicoBoy testen.

KAPITEL 21

Projekt: Ein Spiel Schritt für Schritt

Worum geht's? Du baust ein vollständiges Spiel: „Apfelfang". Äpfel fallen vom Himmel, du fängst sie mit einem Korb. Dabei setzt du fast alles ein, was du bisher gelernt hast: Funktionen, Listen, Canvas, Sprites, Kollisionen und Dateien. Vor allem lernst du, wie man ein größeres Programm in Stufen baut, von denen jede für sich läuft.

21.1 Die Idee und der Plan



Bevor du die erste Zeile schreibst, beschreibst du das Spiel in wenigen Sätzen:

- Unten steht ein **Korb**. Du bewegst ihn mit dem Joystick nach links und rechts.
- Von oben fallen mehrere **Äpfel** gleichzeitig.
- Fängst du einen Apfel, gibt es einen **Punkt**. Fällt ein Apfel ins Gras, verlierst du ein **Leben**.
- Hast du kein Leben mehr, ist das **Spiel zu Ende**. Der beste Punktestand bleibt in einer Datei gespeichert (**Highscore**).

Ein ganzes Spiel auf einmal zu schreiben und erst am Ende zu testen, geht fast immer schief: Es gibt zehn Fehler gleichzeitig, und du weißt nicht, wo du suchen sollst. Teile das Spiel deshalb in **Stufen**. Jede Stufe ist ein lauffähiges Programm, das du testen kannst, bevor die nächste dazukommt:

Stufe	Datei	Was kommt dazu?	Test: Das muss klappen
1	`k21_stufe1.py`	Spielfeld, Korb, Steuerung	Korb fährt links/rechts, bleibt am Rand stehen
2	`k21_stufe2.py`	Äpfel aus einer Liste	drei Äpfel fallen, kommen oben wieder
3	`k21_stufe3.py`	Punkte, Leben, Spielende	Fangen zählt, Verpassen kostet ein Leben, bei 0 ist Schluss
4	`k21_stufe4.py`	Highscore in einer Datei	nach einem neuen Rekord erscheint „Neuer Rekord!"

Mit jeder Stufe wird das Programm länger. Im Buch siehst du deshalb ab Stufe 2 nur die Zeilen, die neu sind. Die vollständigen Dateien liegen bei den Beispielprogrammen.

Das Spiel braucht die Bilder `korb.png` und `apfel.png`. Sie liegen bei den Beispielprogrammen. Lade sie über **Dateien → Bilder verwalten…** hoch, bevor du testest. Fehlt ein Bild, bricht das Programm mit einem `OSError` ab (`ENOENT` heißt: Datei nicht gefunden):

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
OSError: [Errno 2] ENOENT
```

21.2 Spielfeld und Spielfigur

In Stufe 1 entstehen das Spielfeld und der Korb, den du steuern kannst:

k21_stufe1.py

```
# Apfelfang – Stufe 1: Spielfeld und Korb
import pbc
import time

HIMMEL = pbc.rgb(120, 200, 255)
WIESE = pbc.rgb(40, 160, 40)
KORB_Y = 240
TEMPO_KORB = 12

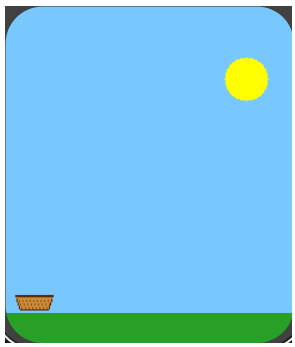

def zeichne_spielfeld(feld):
    feld.fill(HIMMEL)
    feld.ellipse(200, 60, 18, 18, pbc.YELLOW, True)
    feld.fill_rect(0, 255, 240, 25, WIESE)


def bewege_korb(korb):
    # nur bewegen, wenn der Korb danach noch ganz im Bild ist
    if pbc.pressed_left() and korb.x - TEMPO_KORB >= 0:
        korb.set_by(-TEMPO_KORB, 0)
    if pbc.pressed_right() and korb.x + korb.width + TEMPO_KORB <= pbc.WIDTH:
        korb.set_by(TEMPO_KORB, 0)


# Aufbau
feld = pbc.Canvas(240, 280)
zeichne_spielfeld(feld)
pbc.Sprite._background = feld

korb = pbc.sprite("korb.png")
korb.set_to(104, KORB_Y)
korb.show()
```

```
34 # Spielschleife
35 while True:
36     bewege_korb(korb)
37     time.sleep_ms(20)
```



Zeile 5 bis 8: Konstanten. Alle Zahlen, an denen du später drehen willst, stehen oben als Konstanten mit großen Namen: die Farben, die Höhe des Korbs und sein Tempo. Willst du den Korb schneller machen, änderst du nur Zeile 8 und musst nicht im Programm suchen.

Zeile 11 bis 14: `zeichne_spielfeld`. Die Funktion malt Himmel, Sonne und Wiese auf einen Canvas. Sie bekommt den Canvas als Parameter.

Zeile 17 bis 22: `bewege_korb`. Bei gedrücktem Joystick wird der Korb um `TEMPO_KORB` Pixel versetzt, aber nur, wenn er **danach** noch ganz im Bild ist. Zeile 19 rechnet aus, wo die linke Kante nach dem Schritt wäre (`korb.x - TEMPO_KORB`); sie darf nicht kleiner als 0 werden. Rechts zählt die **rechte Kante** des Korbs, also `korb.x + korb.width`. Nach dem Schritt darf sie höchstens bei `pbc.WIDTH` liegen (Zeile 21). Würdest du nur `korb.x > 0` prüfen, könnte der letzte Schritt den Korb ein Stück über den Rand schieben.

Zeile 26 bis 28: der Hintergrund. Der Canvas `feld` wird mit `pbc.Sprite._background = feld` zum Hintergrund hinter allen Sprites.

Zeile 30 bis 32: der Korb. Ein neues Sprite steht bei (0, 0) und ist unsichtbar. Deshalb setzt Zeile 31 den Korb an seinen Platz, und erst `korb.show()` macht ihn sichtbar. Dabei wird der Hintergrund mitgezeichnet.

Zeile 35 bis 37: die Spielschleife. Sie ist hier noch ganz kurz: Tasten lesen und Korb bewegen, kurz warten. Das Hauptprogramm unten liest sich fast wie ein Satz, weil die Einzelheiten in den Funktionen stecken.

Testen: Fahre mit dem Korb ganz nach links und ganz nach rechts. Bleibt er an beiden Rändern stehen? Erst wenn das klappt, geht es weiter.

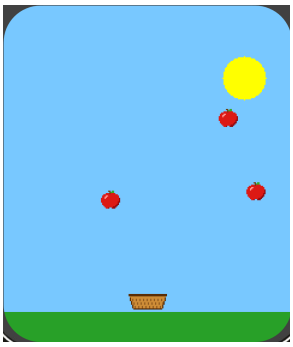
21.3 Gegner aus einer Liste

In Stufe 2 fallen drei Äpfel. Für jeden Apfel gibt es ein eigenes Sprite. Statt drei Variablen `apfel1`, `apfel2`, `apfel3` steckt das Programm alle in eine **Liste** `aepfel`. So ist es egal, ob es drei oder zehn sind: Eine `for`-Schleife behandelt jeden Apfel gleich. Neu in Stufe 2 sind die Konstanten `TEMPO_APFEL = 8` und `ANZAHL_AEPFEL = 3`, `import random` und diese Zeilen:

k21_stufe2.py

```
28 def nach_oben(apfel):
29     # neuer Start oberhalb des Bildschirms, damit die Aepfel versetzt fallen
30     apfel.set_to(random.randint(10, 214), random.randint(-150, -20))
```

```
33 # Aufbau
34 feld = pbc.Canvas(240, 280)
35 zeichne_spielfeld(feld)
36 pbc.Sprite._background = feld
37
38 korb = pbc.sprite("korb.png")
39 korb.set_to(104, KORB_Y)
40 korb.show()
41
42 aepfel = []
43 for i in range(ANZAHL_AEPFEL):
44     apfel = pbc.sprite("apfel.png")
45     nach_oben(apfel)
46     apfel.show()
47     aepfel.append(apfel)
48
49 # Spielschleife
50 while True:
51     bewege_korb(korb)
52     for apfel in aepfel:
53         apfel.set_by(0, TEMPO_APFEL)
54         if apfel.y > 260:
55             nach_oben(apfel)
56     time.sleep_ms(20)
```



Zeile 28 bis 30: `nach_oben`. Die Funktion setzt einen Apfel an eine zufällige Stelle **oberhalb** des Bildschirms: x zwischen 10 und 214 (der Apfel ist 16 Pixel breit), y zwischen −150 und −20. Weil jeder Apfel eine andere Höhe bekommt, fallen sie versetzt herunter und kommen nicht alle gleichzeitig an.

Zeile 42 bis 47: die Liste füllen. Die Schleife läuft `ANZAHL_AEPFEL`-mal. Jedes Mal entsteht ein neues Sprite, das an seinen Start gesetzt, gezeigt und mit `append` an die Liste angehängt wird.

Zeile 52 bis 55: alle Äpfel bewegen. In jedem Durchlauf der Spielschleife fällt jeder Apfel um `TEMPO_APFEL` Pixel. Ist er unten angekommen (`apfel.y > 260`), schickt ihn `nach_oben` wieder an den Anfang. Die Äpfel werden also nie gelöscht und neu angelegt, sondern immer wieder verwendet.

> **Jede Bewegung zeichnet den ganzen Bildschirm.** `set_by` und `set_to` malen bei jedem Aufruf den Hintergrund und **alle** sichtbaren Sprites neu. Im Emulator dauert das jeweils etwa 40 ms, und der Emulator ist ungefähr so schnell wie das Gerät. Mit drei Äpfeln und dem Korb braucht ein Durchlauf der Spielschleife also schon gut 150 ms. Mehr Äpfel machen das Spiel spürbar langsamer. Deshalb fallen die Äpfel 8 Pixel auf einmal und nicht nur 1 Pixel.

Testen: Fallen drei Äpfel an verschiedenen Stellen? Tauchen sie oben wieder auf? Lässt sich der Korb noch steuern? Durch den Korb fallen die Äpfel in dieser Stufe einfach hindurch; das Fangen kommt in der nächsten Stufe.

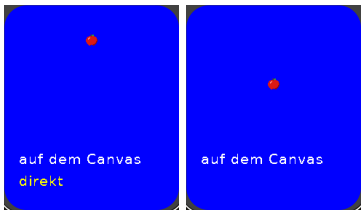
21.4 Punkte, Leben, Spielende

Stufe 3 macht aus der Animation ein Spiel. Dazu braucht es zwei Variablen, `punkte` und `leben`, und eine Anzeige für beide.

Text auf den Canvas, nicht auf den Bildschirm. Sobald Sprites zu sehen sind, malt jede Bewegung den ganzen Bildschirm neu, und zwar aus dem Hintergrund-Canvas. Text, den du direkt mit `pbc.text18` auf den Bildschirm schreibst, ist deshalb beim nächsten `set_by` wieder weg. Text auf dem Canvas bleibt:

k21_text_direkt.py

```
 1 import pbc
 2 import time
 3
 4 feld = pbc.Canvas(240, 280)
 5 feld.fill(pbc.BLUE)
 6 feld.text18("auf dem Canvas", 20, 200, pbc.WHITE)
 7 pbc.Sprite._background = feld
 8
 9 apfel = pbc.sprite("apfel.png")
10 apfel.set_to(112, 40)
11 apfel.show()
12
13 pbc.text18("direkt", 20, 230, pbc.YELLOW)
14 time.sleep(2)
15 apfel.set_by(0, 60)
```



Links siehst du den Bildschirm vor Zeile 15, rechts danach: „direkt“ ist verschwunden, „auf dem Canvas“ ist noch da.

Die Anzeige für Punkte und Leben schreibt das Spiel deshalb auf den Hintergrund-Canvas `feld` (Stufe 3, Zeile 33 bis 37):

k21_stufe3.py

```
33 def zeige_stand(feld, punkte, leben):
34     feld.fill_rect(0, 0, 240, 22, pbc.BLACK)
35     feld.text12(f"Punkte: {punkte}", 10, 5, pbc.WHITE)
36     feld.text12(f"Leben: {leben}", 170, 5, pbc.WHITE)
37     pbc.show(feld)
```

Zeile 34 übermalt den schwarzen Streifen oben, damit der alte Stand verschwindet. Danach schreiben Zeile 35 und 36 den neuen Stand. `pbc.show(feld)` zeigt den Canvas; weil `feld` der Sprite-Hintergrund ist, werden dabei alle Sprites mitgezeichnet. Das kostet Zeit, deshalb ruft das Spiel `zeige_stand` nur auf, wenn sich wirklich etwas ändert.

Und so sieht die Spielschleife jetzt aus:

k21_stufe3.py

```
56 punkte = 0
57 leben = 3
58 zeige_stand(feld, punkte, leben)
59
60 # Spielschleife
61 while leben > 0:
62     bewege_korb(korb)
63     for apfel in aepfel:
64         apfel.set_by(0, TEMPO_APFEL)
65         if apfel.touches(korb):
66             punkte += 1
67             pbc.tone(880, 50)
68             nach_oben(apfel)
69             zeige_stand(feld, punkte, leben)
70         elif apfel.y > 260:
71             leben -= 1
72             pbc.tone(220, 200)
73             nach_oben(apfel)
74             zeige_stand(feld, punkte, leben)
75             if leben == 0:
76                 break
77     time.sleep_ms(20)
78
79 # Spielende
80 korb.hide()
81 for apfel in aepfel:
82     apfel.hide()
83 feld.fill_rect(30, 100, 180, 70, pbc.BLACK)
84 feld.text24("Spielende", 57, 110, pbc.WHITE)
85 feld.text18(f"Punkte: {punkte}", 70, 140, pbc.YELLOW)
86 pbc.show(feld)
```



- **Zeile 56 bis 58:** Startwerte und erste Anzeige.
- **Zeile 61:** Die Schleife läuft nicht mehr endlos, sondern nur, solange noch Leben übrig sind.
- **Zeile 65 bis 69: gefangen.** `apfel.touches(korb)` ist wahr, wenn sich die Rechtecke um Apfel und Korb überlappen. Dann gibt es einen Punkt und einen hohen Ton (`pbc.tone` wartet nicht, das Spiel läuft weiter), und der Apfel startet neu. Ohne `nach_oben` bliebe der Apfel im Korb hängen und würde bei jedem Durchlauf erneut zählen.
- **Zeile 70 bis 76: verpasst.** Ist der Apfel unten, kostet das ein Leben und einen tiefen Ton. `elif` sorgt dafür, dass ein Apfel im selben Durchlauf nicht beides sein kann.
- **Zeile 75 und 76:** Es kann passieren, dass zwei Äpfel im selben Durchlauf unten ankommen. Ohne diese Zeilen würde das Spiel dann „Leben: -1“ anzeigen. `break` beendet die `for`-Schleife sofort, sobald kein Leben mehr übrig ist. Danach prüft Zeile 61 die Bedingung und das Spiel endet.
- **Zeile 80 bis 86: Spielende.** Alle Sprites werden versteckt, dann erscheint ein schwarzer Kasten mit dem Ergebnis auf dem Canvas.

Testen: Fang einen Apfel: Steigen die Punkte um genau 1? Lass einen fallen: Sinkt die Zahl der Leben? Endet das Spiel bei 0 Leben?

21.5 Highscore in einer Datei

Die letzte Stufe merkt sich den besten Punktestand in der Datei `apfel_rekord.txt`, wie in Kapitel 15. Neu sind `import os`, die Konstante `DATEI = "apfel_rekord.txt"` und drei Funktionen:

k21_stufe4.py

```
42 def lade_rekord():
43     if DATEI in os.listdir():
44         with open(DATEI, "r") as f:
45             return int(f.read())
46     return 0
47
48
49 def speichere_rekord(wert):
50     with open(DATEI, "w") as f:
51         f.write(str(wert))
52
53
54 def zeige_ende(feld, punkte, rekord, neu):
55     feld.fill_rect(30, 90, 180, 100, pbc.BLACK)
56     feld.text24("Spielende", 57, 100, pbc.WHITE)
57     feld.text18(f"Punkte: {punkte}", 70, 135, pbc.YELLOW)
58     if neu:
59         feld.text12("Neuer Rekord!", 67, 165, pbc.GREEN)
60     else:
61         feld.text12(f"Rekord: {rekord}", 85, 165, pbc.WHITE)
62     pbc.show(feld)
```

- **`lade_rekord` (Zeile 42 bis 46)** liefert den gespeicherten Rekord als Zahl zurück. Beim allerersten Spiel gibt es die Datei noch nicht. Deshalb prüft Zeile 43 mit `os.listdir()`, ob sie da ist; sonst ist der Rekord 0.
- **`speichere_rekord` (Zeile 49 bis 51)** schreibt eine Zahl in die Datei. Mit `"w"` wird der alte Inhalt ersetzt. `write` braucht einen Text, deshalb `str(wert)`.
- **`zeige_ende` (Zeile 54 bis 62)** zeigt das Ergebnis und, je nach Parameter `neu`, „Neuer Rekord!“ oder den alten Rekord.

Das Ende des Programms benutzt die drei Funktionen:

k21_stufe4.py

```
104 # Spielende und Highscore
105 korb.hide()
106 for apfel in aepfel:
107     apfel.hide()
108 rekord = lade_rekord()
109 neu = punkte > rekord
110 if neu:
111     speichere_rekord(punkte)
112 zeige_ende(feld, punkte, rekord, neu)
113 print("Punkte:", punkte, "Rekord vorher:", rekord)
```



Zeile 109 speichert in `neu`, ob der Rekord geschlagen ist: `True` oder `False`. Nur dann überschreibt Zeile 111 die Datei. Zeile 113 schreibt das Ergebnis zusätzlich ins Terminal; das hilft beim Testen:

Terminal

```
Punkte: 0 Rekord vorher: 0
```

Ob der Rekord wirklich in der Datei steht, prüft das kleine Hilfsprogramm `k21_rekord_lesen.py`. Es liest die Datei und gibt ihren Inhalt aus:

Terminal

```
Noch kein Rekord gespeichert.
```

> **Im Emulator bleibt die Datei nur bis zum Neuladen der Seite.** Lädst du PBC+ Studio neu, ist der Rekord weg, und das nächste Spiel meldet wieder „Neuer Rekord!“. Auf dem Gerät bleibt die Datei dauerhaft gespeichert, auch nach dem Ausschalten.

Zum Testen willst du den Rekord manchmal zurücksetzen. Dafür löscht `os.remove(DATEI)` die Datei (Beispiel: `k21_rekord_loeschen.py`).

21.6 Ideen zum Weiterbauen

Das Spiel ist fertig, aber noch lange nicht zu Ende gedacht. Hier sind Ideen, sortiert von leicht bis schwer. Baue immer nur **eine** Idee auf einmal ein und teste sie, bevor die nächste kommt. Speichere vorher die funktionierende Fassung unter neuem Namen.

Idee	Was du dafür brauchst
Startbild „Druecke A“ vor dem Spiel	`pbc.text24`, `pbc.wait_for_button()` oder `pbc.was_pressed_a()`
die Äpfel werden mit der Zeit schneller	eine Variable `tempo` statt `TEMPO_APFEL`, Rechnen mit `//`

Idee	Was du dafür brauchst
ein fauler Apfel, den man **nicht** fangen darf	zweites Bild, eigenes Sprite, eigene Regel
ein goldener Apfel bringt 5 Punkte	`random.randint`, um selten einen Goldapfel zu schicken
eine Bestenliste mit den drei besten Ergebnissen	Liste, `sorted`, Datei Zeile für Zeile lesen und schreiben
Korb mit dem Bewegungssensor steuern (nur am Gerät)	`pbc.accel_x()` aus Kapitel 20

Übungsaufgaben

Aufgabe 1 ⭐ Was passiert? Beantworte die Fragen zu `k21_stufe4.py`, ohne das Programm zu ändern. Prüfe danach im Emulator.

1. Der Korb steht ganz links, und du drückst weiter nach links. Was passiert, und welche Zeile sorgt dafür?
2. Warum bekommen die Äpfel in `nach_oben` ein negatives y?
3. Du setzt `ANZAHL_AEPFEL = 8`. Was passiert mit dem Tempo des ganzen Spiels? Warum?

Aufgabe 2 ⭐ Ein leichteres Spiel Ändere `k21_stufe4.py` so, dass man 5 Leben hat, der Korb schneller ist und nur zwei Äpfel fallen. Lege für die Zahl der Leben eine neue Konstante `LEBEN_START` an. Welche Zeilen hast du geändert?

Aufgabe 3 ⭐⭐ Finde die Fehler In `k21_aufgabe3.py` stecken zwei Fehler, beide in der Spielschleife (Zeile 60 bis 76). Es erscheint keine Fehlermeldung, aber das Spiel verhält sich falsch. Spiele es, beschreibe, was nicht stimmt, und korrigiere die Fehler. Vergleiche dazu mit dem Programmausschnitt in 21.4.

Aufgabe 4 ⭐⭐ Immer schneller Je mehr Punkte du hast, desto schneller sollen die Äpfel fallen: Alle 5 Punkte 1 Pixel mehr pro Schritt. Tipp: Berechne am Anfang jedes Durchlaufs eine Variable `tempo` aus `TEMPO_APFEL` und `punkte`.

Aufgabe 5 ⭐⭐⭐ **Der faule Apfel** Zusätzlich zu den roten Äpfeln fällt ein fauler Apfel (`faulapfel.png`). Wer ihn fängt, verliert ein Leben; fällt er ins Gras, passiert nichts. Das Spiel muss auch dann richtig enden, wenn er das letzte Leben kostet.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Erst planen, dann in **Stufen** bauen; jede Stufe läuft für sich und wird getestet. Wichtige Zahlen stehen oben als **Konstanten**.
- Funktionen wie `bewege_korb` halten die Spielschleife kurz und lesbar.
- Viele gleiche Gegner stehen in einer **Liste** und werden mit `for` bewegt und wieder nach oben geschickt. `break` beendet eine Schleife sofort.
- Jedes `set_by` / `set_to` zeichnet den ganzen Bildschirm neu. Text gehört auf den Hintergrund-Canvas, angezeigt mit `pbc.show(feld)`.
- Highscore: lesen mit `open(…, "r")`, schreiben mit `open(…, "w")`, vorher mit `os.listdir()` prüfen, ob die Datei existiert.

KAPITEL 22

Dein eigenes Projekt

Worum geht's? Jetzt baust du ein Programm nach deiner eigenen Idee. In diesem Kapitel lernst du, wie man dabei vorgeht: eine Idee finden und planen, in kleinen Schritten bauen, Fehler systematisch finden und das Ergebnis vorstellen. Die Beispiele sind klein, die Arbeitsweise funktioniert aber auch für große Programme.

22.1 Eine Idee finden und planen

Gute Projekte sind **klein genug, um fertig zu werden**, und lassen sich später erweitern. Hier sind Ideen zum Anfangen, sortiert nach Aufwand:

Bereich	eher leicht	mittel	anspruchsvoll
Spiele	Münzwurf-Duell, Zahlenraten mit Display	Reaktionsduell für zwei, Hindernislauf mit Sprites	Snake, Labyrinth
Werkzeuge	Würfel, Stoppuhr	Zufallsnamen für die Klasse, Vokabeltrainer	Taschenrechner mit Joystick
Musik und Licht	Ampel, Metronom	Mini-Klavier, Lichtorgel mit der RGB-LED	Melodie-Rekorder
Grafik	Muster-Generator, Turtle-Kunst	Uhr mit Zeigern	Malprogramm mit dem Joystick
Messen (Gerät)	–	Wasserwaage, Schüttel-Würfel	Schrittzähler

Hast du eine Idee, beantworte vor dem Programmieren drei Fragen:

1. **Was soll das Programm tun?** In ein, zwei Sätzen. Wenn du dafür eine halbe Seite brauchst, ist die Idee zu groß.
2. **Wie bedient man es?** Welche Taste macht was? Was steht auf dem Bildschirm, was in der Konsole?
3. **Was ist die kleinste Fassung, die schon funktioniert?** Alles andere sind Extras für später.

Danach teilst du die Arbeit in **Stufen**, wie in Kapitel 21. Jede Stufe endet mit einem Programm, das läuft. Für einen elektronischen Würfel könnte der Plan so aussehen:

Stufe	Was kommt dazu?	Fertig, wenn …
1	Zufallszahl von 1 bis 6 in der Konsole	bei jedem Start eine Zahl erscheint
2	Würfeln mit Taste A, Zahl auf dem Display	jeder Druck auf A eine neue Zahl zeigt
3	Würfelaugen statt Zahl zeichnen	alle sechs Seiten richtig aussehen
4 (Extra)	Statistik: Wie oft kam welche Zahl?	Taste B die Häufigkeiten ausgibt

> Zeichne vorher auf Karopapier, wie der Bildschirm aussehen soll. Nimm ein Kästchen für 10 Pixel, dann ist der Bildschirm 24 × 28 Kästchen groß. So kannst du die Koordinaten ablesen, statt sie auszuprobieren.

22.2 In kleinen Schritten bauen und testen

Stufe 1 ist winzig. Trotzdem lohnt sich der Test: Läuft `random`, stimmt der Bereich?

k22_schritt1.py

```
# Wuerfel – Schritt 1: eine Zufallszahl
import random

zahl = random.randint(1, 6)
print("Gewuerfelt:", zahl)
```

Terminal

```
Gewuerfelt: 4
```

Starte es mehrmals. Kommen alle Zahlen von 1 bis 6 vor, aber nie 0 oder 7?

Stufe 2 würfelt bei jedem Druck auf Taste A:

k22_schritt2.py

```
# Wuerfel – Schritt 2: mit Taste A wuerfeln, Zahl auf dem Display
import pbc
import random
import time

pbc.clear()
pbc.text18("A = wuerfeln", 52, 20, pbc.WHITE)

while True:
    if pbc.was_pressed_a():
        zahl = random.randint(1, 6)
        print("Gewuerfelt:", zahl)
        pbc.fill_rect(100, 110, 40, 40, pbc.BLACK)
        pbc.text24(str(zahl), 112, 120, pbc.YELLOW)
    time.sleep_ms(20)
```



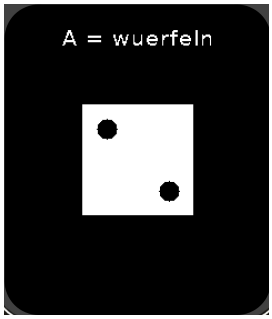
Zeile 12 schreibt jede Zahl zusätzlich ins Terminal. So siehst du auch dann, was passiert, wenn auf dem Display etwas nicht stimmt. Zeile 13 übermalt die alte Zahl mit einem schwarzen Rechteck, bevor Zeile 14 die neue schreibt. `pbc.was_pressed_a()` sorgt dafür, dass ein Druck genau **einmal** würfelt.

Stufe 3 zeichnet Würfelaugen. Das ist der schwierigste Teil, deshalb steckt er in einer eigenen Funktion:

k22_schritt3.py

```
 1 # Wuerfel – Schritt 3: Augen zeichnen
 2 import pbc
 3 import random
 4 import time
 5
 6 # Lage der Augen fuer jede Zahl, gemessen von der Mitte
     (-1, 0 oder 1)
 7 AUGEN = [
 8     [],
 9     [[0, 0]],
10     [[-1, -1], [1, 1]],
11     [[-1, -1], [0, 0], [1, 1]],
12     [[-1, -1], [1, -1], [-1, 1], [1, 1]],
13     [[-1, -1], [1, -1], [0, 0], [-1, 1], [1, 1]],
14     [[-1, -1], [1, -1], [-1, 0], [1, 0], [-1, 1], [1, 1]],
15 ]
16
17
18 def zeichne_wuerfel(zahl):
19     pbc.fill_rect(70, 90, 100, 100, pbc.WHITE)
20     for auge in AUGEN[zahl]:
21         x = 120 + auge[0] * 28
22         y = 140 + auge[1] * 28
23         pbc.ellipse(x, y, 9, 9, pbc.BLACK, True)
24
25
26 pbc.clear()
27 pbc.text18("A = wuerfeln", 52, 20, pbc.WHITE)
28 while True:
29     if pbc.was_pressed_a():
30         zeichne_wuerfel(random.randint(1, 6))
31     time.sleep_ms(20)
```

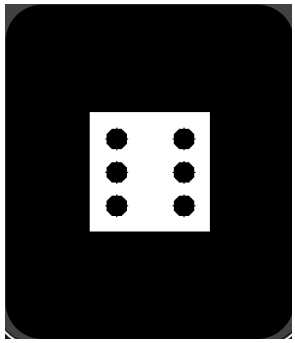
Die Liste `AUGEN` (Zeile 7 bis 15) enthält für jede Zahl eine Liste mit der Lage der Augen, also Listen in Listen wie in Kapitel 14. `[-1, -1]` heißt „links oben“, `[0, 0]` „Mitte“, `[1, 1]` „rechts unten“. Platz 0 bleibt leer, damit `AUGEN[3]` die Augen für die Drei sind. `zeichne_wuerfel` rechnet daraus die Koordinaten aus (Zeile 21 und 22): Die Mitte des Würfels liegt bei (120, 140), und die Augen haben 28 Pixel Abstand.



Jeden Fall testen, nicht nur zufällige. Mit Taste A würfelst du zufällig. Ob wirklich alle sechs Seiten stimmen, siehst du so nur mit Glück. Ein kleines Testprogramm zeichnet deshalb alle Seiten nacheinander:

k22_test_wuerfel.py

```
24 pbc.clear()
25 for zahl in range(1, 7):
26     print("zeichne", zahl)
27     zeichne_wuerfel(zahl)
28     time.sleep(1)
```



Die Funktion `zeichne_wuerfel` ist dieselbe wie in Stufe 3. Die Schleife ruft sie mit 1 bis 6 auf, jeweils eine Sekunde lang. Im Bild siehst du die letzte Seite, die Sechs.

> **Nach jedem Schritt testen.** Wenn plötzlich etwas nicht mehr geht, liegt der Fehler fast immer im Stück, das du zuletzt geschrieben hast. Speichere jede funktionierende Stufe unter einem eigenen Namen (`wuerfel_1.py`, `wuerfel_2.py` …). Dann kannst du jederzeit zurück.

22.3 Fehler finden: `print`, Fehlermeldungen, Programm verkleinern

Fehler gibt es in drei Sorten:

Sorte	Woran erkennst du sie?	Beispiel
Python kann das Programm nicht lesen	`SyntaxError`, keine Zeile läuft	fehlende Klammer, falsche Einrückung
Absturz beim Laufen	Fehlermeldung mitten im Programm	`NameError`, `IndexError`, `TypeError`
Logikfehler	keine Meldung, aber das Ergebnis ist falsch	falsche Formel, falsche Bedingung

Fehlermeldungen lesen. Bei einem Absturz zeigt der Traceback, über welche Funktionsaufrufe Python zur Fehlerstelle gekommen ist:

k22_fehler_index.py

```
1 namen = ["Ada", "Ben", "Cem", "Dora"]
2
3
4 def letzter_name(liste):
5     return liste[len(liste)]
6
7
8 print("Erster:", namen[0])
9 print("Letzter:", letzter_name(namen))
```

Terminal

```
Erster: Ada

Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
  File "<stdin>", line 5, in letzter_name
IndexError: list index out of range
```

Lies wieder von unten: `IndexError` heißt, die Liste hat keinen Platz mit dieser Nummer. Passiert ist es in Zeile 5, in der Funktion `letzter_name`. Die Zeile darüber sagt, wer die Funktion aufgerufen hat: Zeile 9. Die Liste hat vier Namen, also die Plätze 0 bis 3. `len(liste)` ist aber 4. Richtig wäre `liste[len(liste) - 1]`. PBC+ Studio zeigt zusätzlich ein Hinweisfenster „Listen-Position außerhalb".

Mit `print` nachsehen. Ein Logikfehler meldet sich nicht. Dieses Programm soll den Durchschnitt von vier Sprungweiten berechnen:

k22_fehler_mittelwert.py

```
1 # Durchschnitt der Sprungweiten – das Ergebnis stimmt nicht
2 weiten = [3.8, 4.1, 4.5, 3.9]
3
4 summe = 0
5 for weite in weiten:
6     summe = weite
7 durchschnitt = summe / len(weiten)
8 print("Durchschnitt:", durchschnitt)
```

Terminal

```
Durchschnitt: 0.975
```

Ein Durchschnitt unter 1 m bei Weiten um 4 m kann nicht stimmen. Wo geht es schief? Eine `print`-Zeile in der Schleife zeigt, welche Werte die Variablen in jedem Durchlauf haben:

k22_print_suche.py

```
5 for weite in weiten:
6     summe = weite
7     print("weite:", weite, "summe:", summe)  # nur zum Testen
```

Terminal

```
weite: 3.8 summe: 3.8
weite: 4.1 summe: 4.1
weite: 4.5 summe: 4.5
weite: 3.9 summe: 3.9
Durchschnitt: 0.975
```

Jetzt sieht man es sofort: `summe` ist immer nur die aktuelle Weite, statt mitzuwachsen. In Zeile 6 muss `summe += weite` stehen. Nach der Korrektur löschst du die `print`-Zeile wieder:

k22_mittelwert.py

```
5 for weite in weiten:
6     summe += weite
7 durchschnitt = summe / len(weiten)
8 print("Durchschnitt:", round(durchschnitt, 2), "m")
```

Terminal

```
Durchschnitt: 4.07 m
```

`round(durchschnitt, 2)` rundet auf zwei Nachkommastellen. Ohne `round` stünde dort `4.074999999999999`: Kommazahlen werden im Computer nicht ganz genau gespeichert, und die kleine Ungenauigkeit wird bei der Ausgabe sichtbar.

> Schreib in die `print`-Zeile immer den **Namen** der Variable dazu (`print("summe:", summe)`). Sonst stehen im Terminal nur Zahlen, und du weißt nicht, welche zu welcher Variable gehört.

Programm verkleinern. Manchmal versteht man eine Fehlermeldung nicht, weil im Programm zu viel anderes passiert. Dieses Quiz stürzt erst ganz am Ende ab:

k22_quiz.py

```
1 # Hauptstadt-Quiz
2 fragen = ["Frankreich", "Italien", "Spanien"]
3 antworten = ["Paris", "Rom", "Madrid"]
4 punkte = 0
5
6
7 def frage_stellen(land, richtig):
8     eingabe = input(f"Hauptstadt von {land}? ")
9     if eingabe.strip().lower() == richtig.lower():
10         print("Richtig!")
```

```
        return 1
    print("Leider falsch, richtig ist", richtig)
    return 0


for i in range(len(fragen)):
    punkte += frage_stellen(fragen[i], antworten[i])

print("Du hast " + punkte + " von 3 Punkten.")
```

Terminal

```
Hauptstadt von Frankreich? Paris
Richtig!
Hauptstadt von Italien? Rom
Richtig!
Hauptstadt von Spanien? Barcelona
Leider falsch, richtig ist Madrid

Traceback (most recent call last):
  File "<stdin>", line 19, in <module>
TypeError: unsupported types for __add__: 'str', 'int'
```

Dann hilft es, eine **Kopie** des Programms so lange zu verkleinern, bis nur noch das übrig ist, was den Fehler auslöst. Nach jedem Löschen testest du: Kommt der Fehler noch? Hier braucht man am Ende weder Fragen noch Funktion, zwei Zeilen genügen:

k22_typfehler.py

```
punkte = 5
print("Punkte: " + punkte)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported types for __add__: 'str', 'int'
```

Jetzt ist klar: Mit `+` kann man einen Text (`str`) und eine Zahl (`int`) nicht verbinden. Die Lösung im Quiz ist ein f-String: `print(f"Du hast {punkte} von 3 Punkten.")`. Ein verkleinertes Programm eignet sich auch gut, um jemand anderen um Hilfe zu bitten: Es ist kurz, und der Fehler ist sofort zu sehen.

Wenn du feststeckst: Erkläre dein Programm Zeile für Zeile laut, einer Mitschülerin, einem Mitschüler oder auch einem Kuscheltier. Beim Erklären merkt man oft selbst, wo man etwas anderes geschrieben als gemeint hat.

22.4 Vorstellen

Bevor du dein Projekt vorstellst, räumst du das Programm auf, damit andere es lesen können. Eine gute Ordnung ist immer dieselbe: Kopfkommentar, Imports, Konstanten, Funktionen, Hauptprogramm. Diese Vorlage zeigt das Gerüst:

k22_vorlage.py

```
1  # Projekt: Mein Projekt
2  # Von: Vorname Nachname, Klasse 8b
3  # Was es tut: zeigt ein Startbild und laeuft, bis B gedrueckt wird.
4  # Bedienung: A = Start, B = Ende
5  import pbc
6  import time
7
8  # Konstanten
9  FARBE = pbc.CYAN
10
11
12 # Funktionen
13 def zeige_start():
14     pbc.clear()
15     pbc.text24("Mein Projekt", 37, 110, FARBE)
16     pbc.text12("A = Start", 86, 150, pbc.WHITE)
17
18
19 # Hauptprogramm
20 zeige_start()
21 pbc.wait_for_button()
22 pbc.clear()
23 laeuft = True
24 while laeuft:
25     # 1. Tasten lesen
26     if pbc.was_pressed_b():
27         laeuft = False
28     # 2. rechnen und 3. zeichnen kommen hierher
29     time.sleep_ms(20)
30
31 pbc.clear()
32 pbc.text18("Tschuess!", 68, 130, pbc.WHITE)
```



Der Kopfkommentar (Zeile 1 bis 4) sagt, was das Programm ist, wer es geschrieben hat und wie man es bedient. Die Schleife im Hauptprogramm hat die Teile der Spielschleife aus Kapitel 8: Tasten lesen, rechnen, zeichnen, warten.

Für die Vorstellung selbst (etwa 3 bis 5 Minuten) hat sich diese Reihenfolge bewährt:

1. **Was ist es?** Ein Satz zur Idee.
2. **Vorführen.** Im Emulator am Beamer oder auf dem Gerät.
3. **Ein Stück Code erklären**, auf das du stolz bist, zum Beispiel eine Funktion.

4. **Was war schwierig?** Welcher Fehler hat lange gedauert, und wie hast du ihn gefunden?
5. **Was kommt als Nächstes?** Welche Erweiterung würdest du noch bauen?

Checkliste vor dem Vorstellen:

☐ Das Programm läuft im Emulator (und auf dem Gerät) ohne Fehlermeldung.
☐ Ganz oben steht ein Kopfkommentar mit Name, Zweck und Bedienung.
☐ Variablen und Funktionen haben Namen, die sagen, was sie bedeuten.
☐ Wiederholte Programmteile stecken in Funktionen.
☐ Feste Zahlen, die man ändern möchte, stehen als Konstanten oben.
☐ `print`-Zeilen, die nur zur Fehlersuche da waren, sind entfernt.
☐ Auf dem Display stehen keine Umlaute („Tschuess“ statt „Tschüss“).
☐ Die Datei ist gespeichert, mit einem sinnvollen Namen.

Übungsaufgaben

Aufgabe 1 ⭐ Ein Plan für die Stoppuhr Plane eine Stoppuhr: Taste A startet und stoppt, Taste B setzt auf 0 zurück, die Zeit steht in Sekunden mit einer Nachkommastelle auf dem Display. Beantworte die drei Fragen aus 22.1 und schreibe einen Plan mit 3 bis 4 Stufen. Programmieren musst du (noch) nicht.

Aufgabe 2 ⭐⭐ Fehlersuche mit `print` Das Programm soll zählen, an wie vielen Tagen es wärmer als 20 Grad war. Es gibt aber `Warme Tage: 0` aus. Finde den Fehler mit einer `print`-Zeile und korrigiere ihn.

k22_aufgabe2.py

```
# Wie viele Tage waren waermer als 20 Grad?
temperaturen = [18, 22, 25, 19, 21, 30, 17]

for t in temperaturen:
    warme_tage = 0
    if t > 20:
        warme_tage += 1
print("Warme Tage:", warme_tage)
```

Aufgabe 3 ⭐⭐ Verkleinern `k22_aufgabe3.py` zeichnet ein Balkendiagramm, bricht dann aber mit `IndexError` in Zeile 19 ab. Verkleinere eine Kopie des Programms so weit, bis höchstens drei Zeilen übrig sind, die denselben Fehler auslösen. Erkläre damit den Fehler und korrigiere das Diagramm-Programm.

Aufgabe 4 ⭐⭐⭐ Würfel mit Statistik Erweitere `k22_schritt3.py` um Stufe 4 aus dem Plan in 22.1: Das Programm zählt in einer Liste, wie oft jede Zahl gewürfelt wurde. Taste B gibt die Statistik im Terminal aus.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eine Idee in ein, zwei Sätzen beschreiben, die kleinste funktionierende Fassung festlegen, dann in **Stufen** planen.
- Nach jedem kleinen Schritt testen. Funktionierende Stufen unter eigenem Namen speichern.
- Funktionen mit **allen** Fällen testen, nicht nur mit zufälligen.
- Fehlermeldungen von unten lesen: Fehlerart, Zeile, dann wer die Funktion aufgerufen hat.
- Logikfehler mit `print("name:", name)` sichtbar machen, danach die `print`-Zeile wieder entfernen.
- Unverständliche Fehler: eine Kopie des Programms verkleinern, bis nur der Fehler übrig bleibt.
- Vor dem Vorstellen aufräumen: Kopfkommentar, gute Namen, Funktionen, Konstanten.

ZUSATZKAPITEL

Objekte und Klassen

Worum geht's? Du baust dir eigene Bausteine: Klassen. Eine Klasse fasst Daten und die passenden Funktionen zusammen. Aus einer Klasse kannst du beliebig viele Objekte machen, zum Beispiel zehn Bälle, die alle wissen, wo sie sind und wie sie sich bewegen. Dieses Kapitel setzt alle Kapitel 1 bis 22 voraus.

Z.1 Was ist ein Objekt? Du benutzt schon welche

Ohne es zu merken, hast du schon die ganze Zeit mit Objekten gearbeitet:

kZ_objekte.py

```
 1 import pbc
 2
 3 namen = ["Mia", "Tom"]
 4 namen.append("Ela")        # append: eine Methode der Liste
 5 print(namen)
 6
 7 wort = "Python"
 8 print(wort.upper())        # upper: eine Methode des Strings
 9 print(wort.find("t"))      # find: noch eine Methode des Strings
10
11 bild = pbc.Canvas(120, 80)
12 print(bild.width, bild.height)    # Attribute des Canvas
13 bild.fill(pbc.BLUE)               # eine Methode des Canvas
14
15 print(type(namen), type(wort))
```

Terminal

```
['Mia', 'Tom', 'Ela']
PYTHON
2
120 80
<class 'list'> <class 'str'>
```

- Die Liste `namen` ist ein **Objekt**. Sie speichert Werte und bringt Funktionen mit, die zu ihr gehören: `namen.append("Ela")` hängt etwas an **diese** Liste an. Eine Funktion, die zu einem Objekt gehört, heißt **Methode**. Man ruft sie mit einem Punkt auf: `objekt.methode(…)`.
- Auch ein String ist ein Objekt mit Methoden: `upper`, `find`, `replace` …
- Der Canvas `bild` hat zusätzlich **Attribute**: `bild.width` und `bild.height`. Ein Attribut ist ein Wert, der in einem Objekt gespeichert ist. Man schreibt es auch mit Punkt, aber **ohne** Klammern.
- `type()` in Zeile 15 verrät, zu welcher **Klasse** ein Objekt gehört: `namen` gehört zur Klasse `list`, `wort` zur Klasse `str`.

Eine **Klasse** ist der Bauplan, ein **Objekt** ist ein Ding, das nach diesem Bauplan gebaut wurde. Die Klasse `list` beschreibt, was alle Listen können. Jede einzelne Liste in deinem Programm ist ein Objekt dieser Klasse und hat ihre eigenen Werte.

Genauso bei Turtle und Sprite: `t = turtle.Turtle()` baut ein neues Turtle-Objekt. Es merkt sich seine Position, Richtung und Stiftfarbe (Attribute) und kann `forward`, `left`, `penUp` (Methoden). Zwei Turtles laufen unabhängig voneinander, weil jede ihre eigenen Werte hat.

Begriff	Bedeutung	Beispiel
Klasse	Bauplan	`list`, `str`, `pbc.Canvas`
Objekt	ein Ding nach diesem Bauplan	`namen`, `bild`
Attribut	ein Wert im Objekt	`bild.width`
Methode	eine Funktion des Objekts	`namen.append(…)`

Z.2 Eine eigene Klasse: `class`, `__init__`, `self`

In einem Spiel hat jeder Spieler einen Namen, Leben und Punkte. Bisher hättest du dafür drei Listen gebraucht oder viele einzelne Variablen. Mit einer Klasse bekommt jeder Spieler ein eigenes Objekt, das alles zusammenhält:

kZ_spieler.py

```
1  class Spieler:
2      def __init__(self, name, leben):
3          self.name = name
4          self.leben = leben
5          self.punkte = 0
6
7  anna = Spieler("Anna", 3)
8  ben = Spieler("Ben", 5)
9
10 anna.punkte = anna.punkte + 10
11 ben.leben = ben.leben - 1
12
13 print(anna.name, anna.leben, anna.punkte)
14 print(ben.name, ben.leben, ben.punkte)
```

Terminal

```
Anna 3 10
Ben 4 0
```

Zeile 1: `class Spieler:` beginnt den Bauplan. Klassennamen schreibt man in Python mit großem Anfangsbuchstaben. Alles, was eingerückt darunter steht, gehört zur Klasse.

Zeilen 2–5: `__init__` (zwei Unterstriche vorn und hinten) ist eine besondere Methode. Python ruft sie **automatisch** auf, wenn ein neues Objekt gebaut wird. Sie legt die Attribute an und gibt ihnen ihre ersten Werte.

`self` ist immer der erste Parameter einer Methode. Er steht für das Objekt, das gerade gebaut oder benutzt wird. `self.name = name` heißt: „Speichere den Wert des Parameters `name` im Attribut `name` **dieses** Objekts.“ Links steht das Attribut, rechts der Parameter. Dass beide gleich

heißen, ist üblich, aber nicht nötig. `self.punkte` bekommt keinen Parameter: Jeder Spieler startet mit 0 Punkten.

Zeilen 7–8: `Spieler("Anna", 3)` baut ein neues Objekt. Dabei wird `__init__` aufgerufen. Die Werte in den Klammern landen in den Parametern `name` und `leben`, `self` gibt Python von selbst mit. Das fertige Objekt wird in der Variablen `anna` gespeichert.

Zeilen 10–14: Mit `objekt.attribut` liest und änderst du die Attribute. Anna bekommt 10 Punkte, Ben verliert ein Leben. Die Ausgabe zeigt: Jedes Objekt hat seine eigenen Werte.

`self.` vergessen. Innerhalb einer Methode heißt ein Attribut immer `self.attribut`. Ohne `self.` sucht Python eine gewöhnliche Variable und findet keine:

kZ_fehler_self.py

```
1 class Zaehler:
2     def __init__(self):
3         self.stand = 0
4
5     def zeige(self):
6         print("Stand:", stand)
7
8 z = Zaehler()
9 z.zeige()
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
  File "<stdin>", line 6, in zeige
NameError: name 'stand' isn't defined
```

Richtig ist in Zeile 6 `print("Stand:", self.stand)`. Beachte: Die Fehlermeldung nennt zwei Zeilen. Zeile 9 ruft die Methode auf, in Zeile 6 (in `zeige`) passiert der Fehler.

Klassenattribute. Manche Werte gelten für alle Objekte einer Klasse gemeinsam. Sie stehen direkt in der Klasse, nicht in `__init__`, und man erreicht sie über den Klassennamen:

kZ_klassenattribut.py

```
1  class Spieler:
2      MAX_LEBEN = 3     # fuer alle gleich
3      anzahl = 0        # zaehlt die Spieler
4
5      def __init__(self, name):
6          self.name = name
7          self.leben = Spieler.MAX_LEBEN
8          Spieler.anzahl = Spieler.anzahl + 1
9
10 a = Spieler("Anna")
11 b = Spieler("Ben")
12 c = Spieler("Cem")
13 print(Spieler.anzahl, "Spieler")
14 print(a.leben, c.leben, b.MAX_LEBEN)
```

Terminal

```
3 Spieler
3 3 3
```

`MAX_LEBEN` (Zeile 2) ist eine Konstante für alle Spieler, deshalb groß geschrieben. `anzahl` (Zeile 3) gibt es nur **einmal**: Jedes neue Objekt erhöht in Zeile 8 denselben Zähler. Lesen kannst du ein Klassenattribut auch über ein Objekt (`b.MAX_LEBEN` in Zeile 14). Zum Ändern schreibst du immer den Klassennamen davor, wie in Zeile 8.

Z.3 Methoden

Bisher speichern die Objekte nur Werte. Richtig nützlich werden sie durch **Methoden**: Funktionen, die in der Klasse stehen und mit den Attributen des Objekts arbeiten.

kZ_wuerfel.py

```
1  import random
2
3  class Wuerfel:
4      def __init__(self, seiten=6):
5          self.seiten = seiten
6          self.wurf = 1
7          self.anzahl = 0
8
9      def werfen(self):
10         self.wurf = random.randint(1, self.seiten)
11         self.anzahl = self.anzahl + 1
12         return self.wurf
13
14     def ist_hoechster(self):
15         return self.wurf == self.seiten
16
17     def __str__(self):
18         return f"W{self.seiten} zeigt {self.wurf}, Würfe bisher: {self.anzahl}"
19
20 w6 = Wuerfel()
```

```
21 w20 = Wuerfel(20)
22
23 summe = w6.werfen() + w20.werfen()
24 print("Summe:", summe)
25
26 while not w6.ist_hoechster():
27     w6.werfen()
28 print(w6)
29 print(w20)
```

Terminal

```
Summe: 16
W6 zeigt 6, Würfe bisher: 2
W20 zeigt 14, Würfe bisher: 1
```

- **Zeile 4:** Auch `__init__` kann Standardwerte haben (Kapitel 11). `Wuerfel()` baut einen normalen Würfel mit 6 Seiten, `Wuerfel(20)` einen mit 20 Seiten.
- **Zeilen 9–12:** `werfen` ist eine Methode. Sie hat wie jede Methode `self` als ersten Parameter und kann deshalb die Attribute lesen und ändern. Mit `return` liefert sie einen **Rückgabewert** – wie eine Funktion aus Kapitel 12.
- **Zeilen 14–15:** `ist_hoechster` liefert `True` oder `False`.
- **Zeile 23:** Methoden mit Rückgabewert kannst du in Rechnungen benutzen: `w6.werfen() + w20.werfen()`. Beim Aufruf schreibst du `self` **nicht** hin. Python setzt das Objekt vor dem Punkt automatisch ein: Bei `w6.werfen()` ist `self` der Würfel `w6`.
- **Zeilen 26–27:** Die Schleife würfelt mit `w6`, bis eine 6 kommt. Der Würfel zählt dabei selbst mit, wie oft er geworfen wurde.
- **Zeilen 17–18:** `__str__` ist wieder eine besondere Methode. Sie liefert einen Text, der das Objekt beschreibt. `print(w6)` in Zeile 28 ruft sie automatisch auf. Ohne `__str__` würde `print` nur etwas wie `<Wuerfel object at 3ab40>` ausgeben.

Methoden können auch zeichnen. Diese Klasse beschreibt ein Rechteck auf dem Bildschirm:

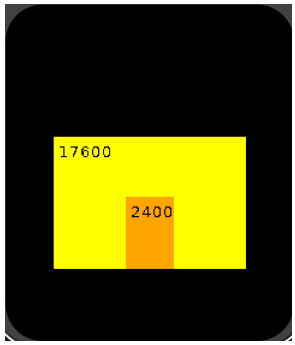
kZ_rechteck.py

```
1  import pbc
2
3  class Rechteck:
4      def __init__(self, x, y, breite, hoehe, farbe):
5          self.x = x
6          self.y = y
7          self.breite = breite
8          self.hoehe = hoehe
9          self.farbe = farbe
10
11     def flaeche(self):
12         return self.breite * self.hoehe
13
14     def enthaelt(self, px, py):
15         drin_x = self.x <= px and px < self.x + self.breite
```

```
16         drin_y = self.y <= py and py < self.y + self.hoehe
17         return drin_x and drin_y
18
19     def zeichne(self):
20         pbc.fill_rect(self.x, self.y, self.breite, self.hoehe, self.farbe)
21         pbc.text12(str(self.flaeche()), self.x + 4, self.y + 4, pbc.BLACK)
22
23 pbc.clear()
24 haus = Rechteck(40, 110, 160, 110, pbc.YELLOW)
25 tuer = Rechteck(100, 160, 40, 60, pbc.ORANGE)
26 haus.zeichne()
27 tuer.zeichne()
28 print("Fläche Haus:", haus.flaeche())
29 print("Ist (110, 200) in der Tür?", tuer.enthaelt(110, 200))
30 print("Ist (50, 120) in der Tür?", tuer.enthaelt(50, 120))
```

Terminal

```
Fläche Haus: 17600
Ist (110, 200) in der Tür? True
Ist (50, 120) in der Tür? False
```



- `flaeche` (Zeilen 11–12) rechnet mit den Attributen.
- `enthaelt` (Zeilen 14–17) bekommt zusätzlich zu `self` zwei Parameter `px` und `py` und prüft, ob dieser Punkt im Rechteck liegt.
- `zeichne` (Zeilen 19–21) ruft in Zeile 21 eine **andere Methode desselben Objekts** auf: `self.flaeche()`. Auch innerhalb der Klasse braucht man dafür `self.` davor.
- Im Hauptprogramm genügen zwei kurze Zeilen, um Haus und Tür zu zeichnen. Wie gezeichnet wird, steht nur einmal in der Klasse.

Z.4 Viele Objekte in einer Liste

Seine ganze Stärke zeigt eine Klasse, wenn es viele Objekte gibt. Objekte kannst du wie Zahlen oder Strings in einer Liste sammeln, durchlaufen, sortieren und auswerten (Kapitel 13 und 14). Hier die Ergebnisse eines 100-m-Laufs:

kZ_lauf.py

```
 1 class Ergebnis:
 2     def __init__(self, name, zeit):
 3         self.name = name
 4         self.zeit = zeit          # Sekunden fuer 100 m
 5
 6     def __str__(self):
 7         return f"{self.name}: {self.zeit} s"
 8
 9 def nach_zeit(ergebnis):
10     return ergebnis.zeit
11
12 lauf = [Ergebnis("Lea", 14.2), Ergebnis("Max", 13.8),
```

```
13          Ergebnis("Ida", 15.1), Ergebnis("Jan", 13.5)]
14 lauf.append(Ergebnis("Ole", 14.9))
15
16 print("Rangliste:")
17 for platz, e in enumerate(sorted(lauf, key=nach_zeit)):
18     print(platz + 1, e)
19
20 schnelle = [e.name for e in lauf if e.zeit < 14.5]
21 print("Unter 14,5 s:", schnelle)
22 print("Bestzeit:", min([e.zeit for e in lauf]))
```

Terminal

```
Rangliste:
1 Jan: 13.5 s
2 Max: 13.8 s
3 Lea: 14.2 s
4 Ole: 14.9 s
5 Ida: 15.1 s
Unter 14,5 s: ['Lea', 'Max', 'Jan']
Bestzeit: 13.5
```

- **Zeilen 12–14:** Die Liste `lauf` enthält fünf `Ergebnis`-Objekte. Jedes hält Name und Zeit zusammen. Mit drei getrennten Listen könnten Namen und Zeiten beim Sortieren durcheinandergeraten. Hier kann das nicht passieren.
- **Zeilen 9–10 und 17:** Wonach soll `sorted` Objekte ordnen? Das sagst du wie in Kapitel 13.4 mit `key=` und dem Namen einer Funktion, **ohne** Klammern. `nach_zeit` liefert für jedes Ergebnis seine Zeit, also wird nach der Zeit sortiert. `enumerate` liefert zu jedem Element eine laufende Nummer (ab 0), deshalb steht in Zeile 18 `platz + 1`.
- **Zeile 18:** `print(platz + 1, e)` benutzt für `e` automatisch `__str__`.
- **Zeile 20:** Eine List Comprehension (Kapitel 14) sammelt die Namen aller Ergebnisse unter 14,5 Sekunden.
- **Zeile 22:** `[e.zeit for e in lauf]` macht aus der Liste von Objekten eine Liste von Zahlen. Darauf funktionieren `min`, `max` und `sum`.

Wann lohnt sich eine Klasse? Wenn mehrere Werte zusammengehören (Name, Zeit; x, y, Richtung, Farbe) und es davon mehrere Exemplare gibt. Für ein kurzes Programm mit einem einzigen Ding sind Variablen und Funktionen oft einfacher.

Im Emulator: Eine Funktion, die du `sorted` mit `key=` übergibst, darf nicht warten (kein `time.sleep`, `input`, `glide_to` …). Auf dem Gerät geht das, im Emulator bricht das Programm dann mit einem `TypeError` ab. Funktionen wie `nach_zeit`, die nur einen Wert zurückgeben, funktionieren überall.

Z.5 Das Spiel aus Kapitel 21 mit Klassen

In Kapitel 21 hast du „Apfelfang“ gebaut: Ein Korb fängt fallende Äpfel. Das Spiel bestand aus Konstanten, Funktionen und einer Spielschleife. Jetzt baust du **dasselbe Spiel** mit Klassen.
Es gibt drei Arten von Dingen im Spiel, also drei Klassen: den `Korb`, den `Apfel` und den `Spielstand` (Punkte und Leben). Die Bilder `korb.png` und `apfel.png` müssen wie in Kapitel 21 hochgeladen sein.

kZ_apfelfang.py

```
# Apfelfang mit Klassen
import pbc
import time
import random
import os

HIMMEL = pbc.rgb(120, 200, 255)
WIESE = pbc.rgb(40, 160, 40)
ANZAHL_AEPFEL = 3
DATEI = "apfel_rekord.txt"


class Korb:
    TEMPO = 12

    def __init__(self):
        self.sprite = pbc.sprite("korb.png")
        self.sprite.set_to(104, 240)
        self.sprite.show()

    def bewege(self):
        s = self.sprite
        if pbc.pressed_left() and s.x - Korb.TEMPO >= 0:
            s.set_by(-Korb.TEMPO, 0)
        if pbc.pressed_right() and s.x + s.width + Korb.TEMPO <= pbc.WIDTH:
            s.set_by(Korb.TEMPO, 0)


class Apfel:
    TEMPO = 8

    def __init__(self):
        self.sprite = pbc.sprite("apfel.png")
        self.nach_oben()
        self.sprite.show()

    def nach_oben(self):
        # neuer Start oberhalb des Bildschirms, versetzt
        self.sprite.set_to(random.randint(10, 214), random.randint(-150, -20))

    def falle(self):
        self.sprite.set_by(0, Apfel.TEMPO)
```

```
44      def gefangen(self, korb):
45          return self.sprite.touches(korb.sprite)
46
47      def im_gras(self):
48          return self.sprite.y > 260
```



Der Korb (Zeilen 13–26). `__init__` legt das Sprite an, setzt es an seine Startposition und zeigt es. Das Sprite wird in einem Attribut gespeichert: `self.sprite`. Der Korb **hat** also ein Sprite. Seine Geschwindigkeit `TEMPO` ist ein Klassenattribut. In `bewege` steht dieselbe Steuerung wie in Kapitel 21. Die Hilfsvariable `s` in Zeile 22 spart nur Schreibarbeit: `s.x` statt `self.sprite.x`.

Der Apfel (Zeilen 29–48). Jeder Apfel legt in `__init__` sein eigenes Sprite an und setzt sich mit `self.nach_oben()` an einen zufälligen Start über dem Bildschirm. Die Methoden heißen so, wie man im Spiel darüber spricht: Der Apfel **fällt**, er wird **gefangen**, er liegt **im Gras**. `gefangen` bekommt den Korb als Parameter und fragt dessen Sprite, ob sich die beiden berühren.

kZ_apfelfang.py

```
51 class Spielstand:
52     def __init__(self, feld):
53         self.feld = feld
54         self.punkte = 0
55         self.leben = 3
56         self.zeige()
57
58     def treffer(self):
59         self.punkte += 1
60         pbc.tone(880, 50)
61         self.zeige()
62
63     def verpasst(self):
64         self.leben -= 1
65         pbc.tone(220, 200)
66         self.zeige()
67
68     def vorbei(self):
69         return self.leben == 0
70
71     def zeige(self):
72         self.feld.fill_rect(0, 0, 240, 22, pbc.BLACK)
73         self.feld.text12(f"Punkte: {self.punkte}", 10, 5, pbc.WHITE)
74         self.feld.text12(f"Leben: {self.leben}", 170, 5, pbc.WHITE)
75         pbc.show(self.feld)
76
77     def zeige_ende(self, rekord, neu):
78         self.feld.fill_rect(30, 90, 180, 100, pbc.BLACK)
79         self.feld.text24("Spielende", 57, 100, pbc.WHITE)
80         self.feld.text18(f"Punkte: {self.punkte}", 70, 135, pbc.YELLOW)
```

```
81          if neu:
82              self.feld.text12("Neuer Rekord!", 67, 165, pbc.GREEN)
83          else:
84              self.feld.text12(f"Rekord: {rekord}", 85, 165, pbc.WHITE)
85          pbc.show(self.feld)
```

Der Spielstand (Zeilen 51–85) hält Punkte und Leben zusammen mit dem Canvas, auf dem sie angezeigt werden. `treffer` und `verpasst` ändern den Stand, spielen einen Ton und zeigen den neuen Stand an. Im Hauptprogramm muss sich niemand mehr darum kümmern, dass nach jeder Änderung neu gezeichnet wird. In Kapitel 21 stand dafür nach jedem `punkte += 1` ein `zeige_stand(feld, punkte, leben)`.

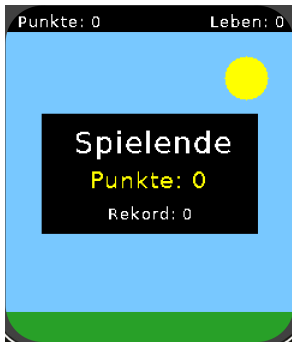
Die Funktionen `lade_rekord` und `speichere_rekord` (Zeilen 88–97) sind genau dieselben wie in Kapitel 21. Nicht alles muss in eine Klasse.

kZ_apfelfang.py

```
100 # Aufbau
101 feld = pbc.Canvas(240, 280)
102 feld.fill(HIMMEL)
103 feld.ellipse(200, 60, 18, 18, pbc.YELLOW, True)
104 feld.fill_rect(0, 255, 240, 25, WIESE)
105 pbc.Sprite._background = feld
106
107 korb = Korb()
108 aepfel = [Apfel() for i in range(ANZAHL_AEPFEL)]
109 stand = Spielstand(feld)
110
111 # Spielschleife
112 while not stand.vorbei():
113     korb.bewege()
114     for apfel in aepfel:
115         apfel.falle()
116         if apfel.gefangen(korb):
117             stand.treffer()
118             apfel.nach_oben()
119         elif apfel.im_gras():
120             stand.verpasst()
121             apfel.nach_oben()
122             if stand.vorbei():
123                 break
124     time.sleep_ms(20)
125
126 # Spielende und Highscore
127 korb.sprite.hide()
128 for apfel in aepfel:
129     apfel.sprite.hide()
130 rekord = lade_rekord()
131 neu = stand.punkte > rekord
132 if neu:
133     speichere_rekord(stand.punkte)
134 stand.zeige_ende(rekord, neu)
135 print("Punkte:", stand.punkte, "Rekord vorher:", rekord)
```

Das Hauptprogramm baut in den Zeilen 107–109 die Objekte: einen Korb, eine Liste mit drei Äpfeln (List Comprehension) und den Spielstand. Die Spielschleife liest sich jetzt fast wie die Spielregeln aus Kapitel 21.1: „Solange das Spiel nicht vorbei ist: Korb bewegen. Jeder Apfel fällt. Wird er gefangen, gibt es einen Treffer. Liegt er im Gras, ist er verpasst."

Was ist übersichtlicher als in Kapitel 21?



- **Zusammengehöriges steht zusammen.** Alles über den Apfel (Bild, Startposition, Fallen, Fangen) steht in der Klasse `Apfel`. In Kapitel 21 war es auf Konstanten, die Funktion `nach_oben` und die Spielschleife verteilt.
- **Weniger Parameter.** `zeige_stand(feld, punkte, leben)` brauchte drei Werte. `stand.zeige()` braucht keinen: Der Spielstand kennt sie selbst.
- **Keine losen Variablen.** `punkte` und `leben` sind keine einzelnen Variablen mehr, die man an jeder Stelle versehentlich ändern kann. Sie ändern sich nur über `treffer` und `verpasst`.
- **Leichter zu erweitern.** Soll es eine zweite Art Apfel geben, änderst du die Klasse `Apfel` und legst andere Objekte an (Aufgabe 6).

Dafür ist das Programm etwas länger geworden: 135 statt 113 Zeilen. Bei einem so kleinen Spiel ist das ein fairer Tausch. Je größer ein Programm wird, desto mehr lohnen sich Klassen.

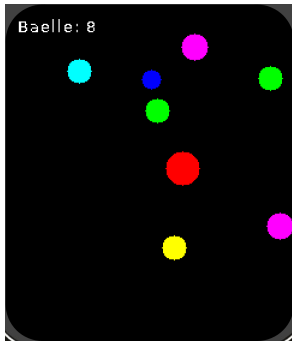
Ausprobieren: Ein Ball-Objekt

Jetzt springen Bälle über den Bildschirm. Jeder Ball ist ein Objekt mit Position, Richtung, Größe und Farbe. Mit **Taste A** kommt ein Ball dazu, **Taste B** nimmt einen weg. Gezeichnet wird auf einen Canvas, damit nichts flackert (Kapitel 17).

kZ_baelle.py

```
 1 import pbc
 2 import random
 3 import time
 4
 5 class Ball:
 6     def __init__(self, farbe):
 7         self.r = random.randint(6, 14)
 8         self.x = random.randint(self.r, pbc.WIDTH - self.r)
 9         self.y = random.randint(self.r, pbc.HEIGHT - self.r)
10         self.dx = random.choice([-3, -2, 2, 3])
11         self.dy = random.choice([-3, -2, 2, 3])
12         self.farbe = farbe
13
14     def bewege(self):
15         self.x = self.x + self.dx
16         self.y = self.y + self.dy
17         # an einer Wand die Richtung umdrehen
18         if self.x < self.r or self.x > pbc.WIDTH - self.r:
19             self.dx = -self.dx
```

```
20          if self.y < self.r or self.y > pbc.HEIGHT - self.r:
21              self.dy = -self.dy
22
23      def zeichne(self, bild):
24          bild.ellipse(self.x, self.y, self.r, self.r, self.farbe, True)
25
26  FARBEN = [pbc.RED, pbc.GREEN, pbc.BLUE, pbc.YELLOW, pbc.CYAN, pbc.MAGENTA]
27  bild = pbc.Canvas(pbc.WIDTH, pbc.HEIGHT)
28  baelle = [Ball(f) for f in FARBEN]
29
30  while True:
31      if pbc.was_pressed_a():
32          baelle.append(Ball(random.choice(FARBEN)))
33      if pbc.was_pressed_b() and len(baelle) > 0:
34          baelle.pop()
35      bild.fill(pbc.BLACK)
36      for ball in baelle:
37          ball.bewege()
38          ball.zeichne(bild)
39      bild.text12(f"Baelle: {len(baelle)}", 10, 10, pbc.WHITE)
40      pbc.show(bild)
41      time.sleep_ms(20)
```



Die Klasse `Ball`. `__init__` (Zeilen 6–12) bekommt nur die Farbe. Alles andere würfelt der Ball selbst aus: einen Radius `r`, eine Startposition, bei der er ganz auf dem Bildschirm liegt, und eine Geschwindigkeit `dx`, `dy` (Pixel pro Bild, nach links/oben negativ). `bewege` (Zeilen 14–21) verschiebt den Ball. Berührt er links oder rechts eine Wand, dreht Zeile 19 das Vorzeichen von `dx` um: Aus 3 wird −3, der Ball fliegt zurück. Zeile 21 macht dasselbe für oben und unten. `zeichne` (Zeilen 23–24) zeichnet nicht direkt auf das Display, sondern auf den Canvas, den es als Parameter `bild` bekommt.

Das Hauptprogramm. Zeile 28 baut mit einer List Comprehension für jede der sechs Farben einen Ball. In der Spielschleife fügt Taste A mit `append` einen neuen Ball an, Taste B entfernt mit `pop` den letzten. Die Zeilen 36–38 sind das Herz des Programms: Für jeden Ball in der Liste wird `bewege` und `zeichne` aufgerufen. Ob es 3 oder 30 Bälle sind, spielt keine Rolle. Jeder Ball weiß selbst, wo er ist und wohin er fliegt.

Zum Verändern:

- Gib der Klasse ein Attribut `stoesse`, das bei jedem Wandkontakt um 1 wächst, und zeichne die Zahl in den Ball.
- Lass die Bälle langsam nach unten fallen: In `bewege` wird `dy` bei jedem Aufruf etwas größer (Schwerkraft). Was musst du am Boden ändern?

Übungsaufgaben

Aufgabe 1 ⭐ Zwei Ampeln Was gibt das Programm aus? Überlege zuerst, dann teste.

kZ_aufgabe1.py

```python
class Ampel:
    def __init__(self):
        self.phasen = ["rot", "rot-gelb", "gruen", "gelb"]
        self.nr = 0

    def weiter(self):
        self.nr = (self.nr + 1) % len(self.phasen)

    def farbe(self):
        return self.phasen[self.nr]

a = Ampel()
b = Ampel()
a.weiter()
a.weiter()
b.weiter()
print(a.farbe(), b.farbe())
for i in range(3):
    a.weiter()
print(a.farbe(), a.nr)
```

Aufgabe 2 ⭐ Finde die Fehler Das Programm soll die mittlere Temperatur und den höchsten Messwert ausgeben. Es enthält zwei Fehler. Teste, lies die Fehlermeldung, korrigiere und teste erneut.

kZ_aufgabe2.py

```python
class Thermometer:
    def __init__(self, ort):
        self.ort = ort
        self.werte = []

    def messen(self, wert):
        self.werte.append(wert)

    def mittel():
        return sum(self.werte) / len(self.werte)

hof = Thermometer("Schulhof")
hof.messen(12)
hof.messen(15)
hof.messen(18)
print(hof.ort, hof.mittel())
print("Hoechster Wert:", max(werte))
```

Aufgabe 3 ⭐⭐ Rechtecke erweitern Erweitere die Klasse `Rechteck` aus Z.3 um die Methoden `umfang()` und `ist_quadrat()`. Lege drei Rechtecke in einer Liste an und zeichne sie in einer Schleife. Quadrate bekommen zusätzlich einen weißen Rahmen. Gib für jedes Rechteck Umfang und „Quadrat ja/nein“ in der Konsole aus.

Aufgabe 4 ⭐⭐ Das Sparschwein Schreibe eine Klasse `Konto` mit den Attributen `besitzer` und `stand` (Start 0) und den Methoden `einzahlen(betrag)` und `abheben(betrag)`. `abheben` soll nur abheben, wenn genug Geld da ist, und `True` oder `False` zurückgeben. Mit `__str__` soll `print(konto)` zum Beispiel „Konto von Jonas: 35 Euro“ zeigen.

Aufgabe 5 ⭐⭐⭐ Bälle, die sich berühren Erweitere das Ball-Programm: Gib der Klasse `Ball` eine Methode `beruehrt(anderer)`, die `True` liefert, wenn sich zwei Bälle überlappen. Tipp: Zwei Kreise überlappen sich, wenn der Abstand ihrer Mittelpunkte kleiner ist als die Summe der Radien (Kapitel 19.4). Bälle, die gerade einen anderen berühren, werden weiß gezeichnet.

Aufgabe 6 ⭐⭐⭐ Der faule Apfel mit Klassen Erweitere `kZ_apfelfang.py` aus Z.5: Zusätzlich zu den drei roten Äpfeln fällt ein fauler Apfel (`faulapfel.png`). Wer ihn fängt, verliert ein Leben. Fällt er ins Gras, passiert nichts. Tipp: Gib `Apfel.__init__` die Parameter `bild` und `faul` (Standardwert `False`). Vergleiche mit Aufgabe 5 aus Kapitel 21: Welche Lösung war einfacher?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eine **Klasse** ist ein Bauplan, ein **Objekt** ein Ding nach diesem Bauplan. Objekte haben **Attribute** (Werte) und **Methoden** (Funktionen).
- `class Name:` beginnt eine Klasse. `Name(…)` baut ein neues Objekt.
- `__init__(self, …)` wird beim Bauen automatisch aufgerufen und legt die Attribute an: `self.x = x`.
- Jede Methode hat `self` als ersten Parameter. Innerhalb der Klasse heißen Attribute und Methoden immer `self.…`. Beim Aufruf `objekt.methode()` setzt Python `self` selbst ein.
- `__str__` liefert den Text für `print(objekt)`.
- Klassenattribute stehen direkt in der Klasse und gelten für alle Objekte.
- Viele Objekte sammelt man in einer Liste; `sorted(…, key=funktion)` sortiert sie nach einem Attribut.

ANHANG A

Befehlsübersicht

Worum geht's? Alle Befehle aus diesem Buch auf einen Blick, zum Nachschlagen beim Programmieren. Die Spalte **Kap.** nennt den Abschnitt, in dem der Befehl erklärt wird. Ein Strich (–) heißt: Der Befehl funktioniert, wird im Buch aber nicht eigens erklärt.

Python-Grundlagen

Datentypen und Umwandlung

Typ / Befehl	Bedeutung	Beispiel	Kap.
`int`	ganze Zahl	`42`, `-7`	4.2
`float`	Kommazahl (mit Punkt!)	`3.14`	4.2
`str`	Text (String)	`"Hallo"`	2.2, 4.2
`bool`	Wahrheitswert	`True`, `False`	7.3
`list`	Liste	`[3, 1, 2]`	13.1
`None`	„nichts", z. B. Ergebnis einer Funktion ohne `return`		12.1
`int(x)`, `float(x)`, `str(x)`	umwandeln	`int("12")` → `12`	3.3
`type(x)`	Typ eines Werts	`type(3.5)`	4.2
`f"…{x}…"`	Text mit Werten (f-String)	`f"Punkte: {p}"`	4.6

Operatoren

Operator	Bedeutung	Beispiel	Kap.
`+ - * /`	Grundrechenarten (`/` liefert immer `float`)	`7 / 2` → `3.5`	4.3
`//`, `%`	ganzzahlig teilen, Rest	`7 // 2` → `3`, `7 % 2` → `1`	4.3
`**`	hoch	`2 ** 10` → `1024`	4.3
`=`, `+=`, `-=`	zuweisen, erhöhen, verringern	`x += 1`	4.1, 4.4
`== != < <= > >=`	vergleichen	`x == 5`	7.2

Operator	Bedeutung	Beispiel	Kap.
`and`, `or`, `not`	verknüpfen	`x > 0 and x < 10`	7.3
`in`	ist enthalten in Text oder Liste	`"a" in "Hallo"`	10.3, 13.4
`+`, `*` bei Text/Listen	aneinanderhängen, wiederholen	`"ab" * 3`, `[0] * 5`	10, 13

Kontrollstrukturen

Befehl	Bedeutung	Kap.
`# …`	Kommentar	2.5
`import modul`	Modul laden (ganz oben)	2.1
`if …:` / `elif …:` / `else:`	Entscheidung	7.1
`for i in range(n):`	n-mal wiederholen, i = 0 … n−1	6.2
`range(a, b)`, `range(a, b, schritt)`	Zahlen von a bis b−1 (mit Schritt)	6.3
`for x in liste:`	jedes Element durchlaufen (auch Text, Datei)	10.4, 13.3, 15.2
`while True:`	Endlosschleife	6.5
`while bedingung:`	wiederholen, solange die Bedingung wahr ist	7.5
`break`	Schleife sofort verlassen	7.5
`with open(…) as d:`	Datei öffnen, automatisch schließen	15.3
`class Name:`	eigene Klasse	Z.2

Funktionen

Befehl	Bedeutung	Kap.
`def name(a, b):`	Funktion mit Parametern definieren	11.1, 11.2
`def name(a, b=10):`	Parameter mit Standardwert	11.3
`return wert`	Ergebnis zurückgeben, Funktion beenden	12.1
`return a, b` / `x, y = f()`	mehrere Rückgabewerte	12.3
`global name`	globale Variable in der Funktion ändern	12.4
Funktion ruft sich selbst auf	Rekursion, braucht Abbruchbedingung	16.6

Eingebaute Funktionen

Befehl	Bedeutung	Kap.
`print(a, b, …)`	in die Konsole schreiben	3.1
`input("Frage")`	Eingabe lesen (liefert immer Text)	3.2
`len(x)`	Länge von Text oder Liste	10.1, 13.4
`min`, `max`, `sum`	kleinster, größter Wert, Summe	13.4
`sorted(l)`, `sorted(l, reverse=True)`, `sorted(l, key=f)`	sortierte Kopie	13.4
`round(x)`, `round(x, 1)`	runden (auf Stellen)	12
`abs(x)`	Betrag (ohne Vorzeichen)	–
`ord("A")` → `65`, `chr(65)` → `"A"`	Zeichen ↔ Zahl	10.5
`enumerate(l)`	Paare (Nummer, Element)	Z.4
`zip(l1, l2)`	zwei Listen paarweise durchlaufen	–
`isinstance(x, int)`	hat x diesen Typ?	–

Listen

Befehl	Bedeutung	Kap.
`l[0]`, `l[-1]`, `l[i] = x`	Element lesen (erstes, letztes), ändern	13.1
`l.append(x)`	hinten anhängen	13.2
`l.insert(i, x)`	an Stelle i einfügen	13.2
`l.remove(x)`	ersten Wert x entfernen	13.2
`l.pop()`, `l.pop(i)`	letztes / i-tes Element entfernen und liefern	13.2
`l.sort()`, `l.reverse()`	Liste selbst sortieren, umdrehen	13.4
`l.count(x)`, `l.index(x)`	Anzahl von x, Stelle von x	13.4
`[[0, 0], [1, 2]]`, `l[i][j]`	Listen in Listen	14.2
`[x * 2 for x in l if x > 0]`	List Comprehension	14.4

Strings (Texte)

Befehl	Bedeutung	Kap.
`s[i]`, `s[a:b]`	Zeichen, Teilstück	10.1, 10.2
`s.upper()`, `s.lower()`	groß, klein	10.3
`s.replace(alt, neu)`	ersetzen	10.3
`s.strip()`	Leerzeichen und `\n` am Rand entfernen	15.2
`s.split(",")`	in Liste zerlegen	–
`",".join(liste)`	Liste zu Text verbinden	–
`s.find(t)`	Stelle von t (−1 = nicht da)	–
`s.startswith(t)`, `s.endswith(t)`	beginnt / endet mit t?	–
`s.isdigit()`	nur Ziffern?	Anhang B

Dateien und Modul `os`

Befehl	Bedeutung	Kap.
`d = open("name.txt", "w")`	zum Schreiben öffnen (neu, alter Inhalt weg)	15.1
`open(…, "a")`, `open(…, "r")`	anhängen, lesen	15.1
`d.write(text)`	schreiben (nur Text; `"\n"` für neue Zeile)	15.1
`d.read()`, `d.readline()`	alles / eine Zeile lesen	15.1, 15.2
`d.close()`	schließen	15.1
`import os` · `os.listdir()`	Dateinamen als Liste	15.4
`os.remove("name.txt")`	Datei löschen	15.4

Modul `pbc`

`import pbc` · Bildschirm 240 × 280 Pixel, (0, 0) oben links, y nach unten.

Display, Farben, Text

Befehl	Bedeutung	Kap.
`pbc.clear()`, `pbc.clear(farbe)`, `pbc.fill(farbe)`	Bildschirm löschen / füllen	2.2
`pbc.WIDTH`, `pbc.HEIGHT`	240, 280	5.4
`pbc.BLACK WHITE RED GREEN BLUE YELLOW CYAN MAGENTA ORANGE PURPLE GRAY`	Farbkonstanten	2.2, 5.3
`pbc.rgb(r, g, b)`	eigene Farbe, je 0–255	5.3

Befehl	Bedeutung	Kap.
`pbc.pixel(x, y, farbe)`	ein Pixel	5.1
`pbc.line(x1, y1, x2, y2, farbe)`	Linie	5.1
`pbc.hline(x, y, laenge, farbe)`, `pbc.vline(…)`	waagrechte / senkrechte Linie	5.1
`pbc.rect(x, y, b, h, farbe)`, `pbc.fill_rect(…)`	Rechteck: Umriss / gefüllt (auch `rect(…, fill=True)`)	5.2
`pbc.ellipse(mx, my, rx, ry, farbe)`, `…, fill=True`	Ellipse / Kreis um den Mittelpunkt	5.2
`pbc.text(s, x, y, farbe)`	8×8-Schrift, alle Zeichen gleich breit	2.4
`pbc.text12/text18/text24(s, x, y, farbe)`	Schrift 12/18/24 px; optional `bg=farbe`	2.4
`pbc.text12_width(s)` (ebenso 18, 24)	Breite eines Texts in Pixeln	5.4
`pbc.delay(ms)`	warten (wie `time.sleep_ms`)	6.1

Tasten und Joystick

Befehl	Bedeutung	Kap.
`pbc.pressed_up/down/left/right/center/a/b()`	`True`, **solange** gedrückt	8.1
`pbc.was_pressed_up/…/a/b()`	`True` genau **einmal** pro Druck	8.2
`pbc.any_pressed()`	irgendeine Taste gerade gedrückt?	–
`pbc.wait_for_button()`	warten, bis eine Taste gedrückt und losgelassen wurde	8.3

LEDs, Ton, Sensor

Befehl	Bedeutung	Kap.
`pbc.led_red(True)`, `led_yellow`, `led_green`	Status-LED an (`False` = aus)	9.1
`pbc.leds_off()`	alle Status-LEDs aus	9.1
`pbc.rgb_led(r, g, b)`, `pbc.rgb_led_off()`	RGB-LED, je 0–255	9.2
`pbc.tone(frequenz)`, `pbc.tone(frequenz, ms)`	Ton in Hz; **wartet nicht**	9.3
`pbc.tone_off()`	Ton aus	9.3

Befehl	Bedeutung	Kap.
`pbc.accel_x()`, `accel_y()`, `accel_z()`	Beschleunigung in m/s² (Emulator: 0, 0, 9.81)	20.2
`pbc.accel_x_g()`, `accel_y_g()`, `accel_z_g()`	dasselbe in g	20.2
`pbc.battery_voltage()`, `pbc.is_on_usb()`	Batteriespannung, am USB?	–

Canvas (Bild im Speicher)

Befehl	Bedeutung	Kap.
`c = pbc.Canvas(b, h)`	neuer Canvas (ganzer Bildschirm: rund 134 KB)	17.1
`c.fill`, `c.pixel`, `c.line`, `c.hline`, `c.vline`, `c.rect`, `c.fill_rect`, `c.ellipse`, `c.text`, `c.text12/18/24`	zeichnen wie mit `pbc.…`	17.2
`c.pixel(x, y)`	Farbe eines Pixels lesen	17.2
`c.width`, `c.height`	Größe	17.1
`pbc.show(c)`, `pbc.show(c, x, y)`	Canvas auf dem Display zeigen	17.2
`bild = pbc.load_image("datei.png")`	PNG laden → Canvas (Magenta = durchsichtig)	17.3
`c.blit(quelle, x, y)`	anderen Canvas hineinkopieren	17.3
`quelle.stamp_on(ziel, x, y)`	mit Durchsichtigkeit stempeln	17.3

Sprites

Befehl	Bedeutung	Kap.
`s = pbc.sprite("bild.png")`	Sprite anlegen (bei (0, 0), unsichtbar)	18.1
`s.show()`, `s.hide()`	zeigen, verstecken	18.2
`s.set_to(x, y)`, `s.set_by(dx, dy)`	setzen, versetzen	18.3
`s.glide_to(x, y, ms)`, `s.glide_by(dx, dy, ms)`	gleiten (wartet, bis fertig)	18.3
`s.x`, `s.y`, `s.width`, `s.height`	Position und Größe lesen	18.4
`pbc.Sprite._background = c`	Hintergrund-Canvas hinter den Sprites	18.5
`s.touches(anderes)`	berührt ein anderes Sprite?	19.1
`s.touches_edge()`	berührt den Rand?	19.2
`s.on_color(farbe)`	steht auf dieser Farbe des Hintergrunds?	19.3

Modul `turtle`

`import turtle` · (0, 0) in der Mitte, y nach oben. Richtung: 0 = oben, 90 = rechts, 180 = unten, 270 = links.

Befehl	Bedeutung	Kap.
`t = turtle.Turtle()`	neue Turtle, Bildschirm schwarz, Blick nach oben	16.1
`t.forward(d)`, `t.back(d)`	vorwärts, rückwärts	16.2
`t.left(w)`, `t.right(w)`	um w Grad drehen	16.2
`t.setDirection(w)`	Richtung setzen	16.2
`t.setPosition(x, y)`	hinspringen, zeichnet **nicht**	16.2
`t.moveTo(x, y)`	hinlaufen, zeichnet mit Stift unten	16.2
`t.home()`	zur Mitte springen	16.2
`t.penUp()`, `t.penDown()`	Stift heben, senken	16.3
`t.setPenColor(farbe)`	Farbe (Konstante oder `"red"`, `"blue"` ...)	16.3
`t.speed(s)`	0 = sofort, 1 langsam ... 10 schnell	16.3
`t.hide()`, `t.show()`	Turtle verstecken, zeigen	16.3
`t.clear()`, `t.clear(farbe)`	Zeichenfläche löschen	16.3
`t.on_color(farbe)`	diese Farbe direkt vor der Turtle?	16.5
`turtle.Turtle(canvas=c)`	auf einen eigenen Canvas zeichnen	–

Module `time`, `random`, `math`

Befehl	Bedeutung	Kap.
`time.sleep(s)`, `time.sleep_ms(ms)`	warten (Sekunden, Millisekunden)	6.1
`time.ticks_ms()`	Millisekunden-Zähler	8.5
`time.ticks_diff(neu, alt)`	Zeitunterschied in ms	8.5
`random.randint(a, b)`	ganze Zufallszahl von a bis b	4.5
`random.random()`	Kommazahl von 0 bis unter 1	4.5
`random.choice(liste)`	zufälliges Element	–
`math.sqrt(x)`	Wurzel	19.4
`math.pi`	3.14159...	–

Befehl	Bedeutung	Kap.
`math.sin(w)`, `math.cos(w)`	Sinus, Kosinus (w im Bogenmaß)	–
`math.radians(grad)`	Grad → Bogenmaß	–
`math.floor(x)`, `math.ceil(x)`	abrunden, aufrunden	–

Klassen (Zusatzkapitel)

Befehl	Bedeutung	Kap.
`class Name:`	Klasse (Bauplan)	Z.2
`def __init__(self, …):`	wird beim Bauen aufgerufen, legt Attribute an	Z.2
`self.x = x`	Attribut des Objekts	Z.2
`obj = Name(…)`	Objekt bauen	Z.2
`def methode(self, …):` / `obj.methode(…)`	Methode definieren / aufrufen	Z.3
`def __str__(self):`	Text für `print(obj)`	Z.3
Wert direkt in der Klasse	Klassenattribut, für alle Objekte gleich	Z.2

ANHANG B

Fehlermeldungen verstehen

Worum geht's? Hier findest du die häufigsten Fehlermeldungen, jeweils mit einem kleinen Programm, das den Fehler macht, der echten Meldung aus PBC+ Studio, einer Erklärung und der Korrektur. Schlag hier nach, wenn dir eine Meldung nichts sagt.

So liest du eine Fehlermeldung

Eine Fehlermeldung (ein **Traceback**) liest du am besten von **unten nach oben** (Kapitel 2.6):

1. Die **letzte Zeile** nennt die Fehlerart (`NameError`, `TypeError` …) und eine kurze Beschreibung auf Englisch.
2. Darüber steht, **in welcher Zeile** der Fehler passiert ist: `line 4`. Die Nummer ist die Zeilennummer links im Editor.
3. Passiert der Fehler in einer Funktion, stehen dort mehrere `File`-Zeilen: oben die Zeile, die die Funktion aufgerufen hat, unten die Zeile **in** der Funktion (`in treffer`). Der Fehler liegt meistens in der untersten.

Die Beschreibungen in MicroPython klingen manchmal etwas anders als in Python auf dem PC, zum Beispiel `isn't defined` statt `is not defined`. Die Fehlerart ist aber dieselbe.

Zusätzlich öffnet PBC+ Studio ein **Hinweisfenster** mit einer Erklärung auf Deutsch. Die Überschriften findest du in der Tabelle am Ende dieses Anhangs.

Zwei Arten von Fehlern: Ein `SyntaxError` (auch `IndentationError`) bedeutet, dass Python das Programm gar nicht lesen kann. Dann läuft **keine einzige** Zeile. Alle anderen Fehler passieren **während** des Laufs: Alles vor der Fehlerzeile wurde schon ausgeführt.

`SyntaxError` – Python versteht eine Zeile nicht

kB_syntax.py

```
1 alter = 14
2 if alter >= 12
3     print("Du darfst mitspielen.")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2
SyntaxError: invalid syntax
```

Am Ende von Zeile 2 fehlt der **Doppelpunkt**. Hinter `if`, `elif`, `else`, `for`, `while`, `def` und `class` steht immer ein `:`.

```
if alter >= 12:
```

Andere häufige Ursachen: eine Klammer oder ein Anführungszeichen fehlt.

kB_syntax_text.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Hallo, 20, 20, pbc.WHITE)
5 pbc.text12("Viel Spass!", 20, 60, pbc.WHITE)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4
SyntaxError: invalid syntax
```

Nach `"Hallo` fehlt das schließende Anführungszeichen. Python hält deshalb den ganzen Rest der Zeile für Text. Richtig: `pbc.text24("Hallo", 20, 20, pbc.WHITE)`. Bei einem `SyntaxError` liegt der Fehler in der genannten Zeile **oder kurz davor**. Typisch ist auch `=` statt `==` in einem Vergleich: `if x = 5:`.

`IndentationError` – die Einrückung stimmt nicht

kB_einrueckung.py

```
1 for i in range(3):
2     print("Runde", i)
3      print("geschafft")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 3
IndentationError: unexpected indent
```

Zeile 3 ist um ein Leerzeichen **zu weit** eingerückt („unexpected indent“: unerwartete Einrückung). Zeilen, die zusammengehören, müssen genau gleich weit eingerückt sein (Kapitel 7.4). Soll „geschafft“ nur einmal am Ende erscheinen, gehört die Zeile ganz nach links:

```
for i in range(3):
    print("Runde", i)
print("geschafft")
```

Fehlt die Einrückung ganz, meldet MicroPython einen `SyntaxError`:

kB_einrueckung_fehlt.py

```
1 punkte = 7
2 if punkte > 5:
3 print("Super!")
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 3
SyntaxError: invalid syntax
```

Nach `if …:` muss mindestens eine eingerückte Zeile folgen. Rücke Zeile 3 mit 4 Leerzeichen ein.

`NameError` – einen Namen gibt es nicht

kB_name.py

```
1 punkte = 0
2 for runde in range(3):
3     punkte = punkte + 10
4 print("Ergebnis:", punkt)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'punkt' isn't defined
```

Die Variable heißt `punkte`, in Zeile 4 steht aber `punkt`. Python unterscheidet jeden Buchstaben und auch Groß- und Kleinschreibung. Weitere Ursachen: `import pbc` vergessen (Kapitel 2.1), eine Variable benutzt, bevor ihr ein Wert zugewiesen wurde, oder eine Funktion aufgerufen, bevor sie mit `def` definiert wurde.

Eine besondere Form tritt in Funktionen auf:

kB_name_lokal.py

```
1 punkte = 0
2
3 def treffer():
4     punkte = punkte + 1
5
6 treffer()
7 print(punkte)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
  File "<stdin>", line 4, in treffer
NameError: local variable referenced before assignment
```

„local variable referenced before assignment“ heißt: Eine **lokale** Variable wird gelesen, bevor sie einen Wert hat. Weil `treffer` der Variablen `punkte` etwas zuweist, hält Python sie für eine lokale Variable der Funktion. Die hat aber noch keinen Wert. Soll die Funktion die **globale** Variable ändern, brauchst du `global` (Kapitel 12.4):

```
def treffer():
    global punkte
    punkte = punkte + 1
```

`AttributeError` – eine Methode oder ein Attribut gibt es nicht

kB_attribut.py

```
1 farben = ["rot", "gruen"]
2 farben.add("blau")
3 print(farben)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'list' object has no attribute 'add'
```

Listen haben keine Methode `add`. Anhängen heißt `append`: `farben.append("blau")`. Dieselbe Meldung kommt bei Tippfehlern in Modulbefehlen:

kB_modul.py

```
1 import random
2
3 print(random.randInt(1, 6))
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
AttributeError: module 'random' has no attribute 'randInt'
```

Richtig ist `random.randint` (alles klein). Auch `pbc.white` statt `pbc.WHITE` führt zu dieser Meldung. Tipp: Lies die Meldung genau. `'list' object` heißt: Der Wert vor dem Punkt ist eine Liste. Hast du dort etwas anderes erwartet, liegt der Fehler weiter oben.

`ImportError` – ein Modul gibt es nicht

kB_import.py

```
1 import pcb
2
3 pcb.clear()
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: no module named 'pcb'
```

Das Modul heißt `pbc`, nicht `pcb`. Die Module in diesem Buch sind `pbc`, `turtle`, `time`, `random`, `math` und `os`.

`TypeError` – ein Wert hat den falschen Typ

Ein `TypeError` entsteht, wenn ein Befehl mit einem Wert nichts anfangen kann. Der häufigste Fall: Text und Zahl mit `+` verbinden.

kB_typ_text.py

```
1 punkte = 42
2 print("Punkte: " + punkte)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported types for __add__: 'str', 'int'
```

„unsupported types for __add__: ‚str', ‚int'" heißt: `+` funktioniert nicht mit einem String und einer ganzen Zahl. Lösungen (Kapitel 3 und 4):

```
print("Punkte: " + str(punkte))
print("Punkte:", punkte)
print(f"Punkte: {punkte}")
```

Besonders tückisch ist das bei `input`, denn `input` liefert **immer** einen String, auch wenn eine Zahl eingetippt wird:

kB_typ_input.py

```
1 alter = input("Wie alt bist du? ")
2 print("Nächstes Jahr bist du", alter + 1)
```

Terminal

```
Wie alt bist du? 13

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported types for __add__: 'str', 'int'
```

Richtig: `alter = int(input("Wie alt bist du? "))` (Kapitel 3.3).

Eine Zahl auf das Display schreiben: Die `pbc.text`-Befehle erwarten einen String.

kB_typ_display.py

```
1 import pbc
2
3 punkte = 42
4 pbc.clear()
5 pbc.text24(punkte, 20, 20, pbc.WHITE)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError: can't convert 'int' object to str implicitly
```

Richtig: `pbc.text24(str(punkte), 20, 20, pbc.WHITE)` oder mit f-String `pbc.text24(f"Punkte: {punkte}", 20, 20, pbc.WHITE)`.

Falsche Anzahl von Argumenten:

kB_typ_argumente.py

```
1 def flaeche(breite, hoehe):
2     return breite * hoehe
3
4 print(flaeche(4))
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: flaeche() takes 2 positional arguments but 1 were given
```

`flaeche` hat zwei Parameter, beim Aufruf steht aber nur ein Wert in den Klammern. Richtig: `flaeche(4, 3)`. Bei Methoden zählt `self` mit (siehe Zusatzkapitel).

Eine Funktion ohne `return`:

kB_typ_none.py

```
1 def verdopple(zahl):
2     print(zahl * 2)
3
4 ergebnis = verdopple(5) + 1
5 print(ergebnis)
```

Terminal

```
10

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: unsupported types for __add__: 'NoneType', 'int'
```

`verdopple` **zeigt** das Ergebnis mit `print`, gibt es aber nicht zurück. Eine Funktion ohne `return` liefert den besonderen Wert `None` („nichts“), und mit `None` kann man nicht rechnen. Richtig: `return zahl * 2` statt `print(zahl * 2)` (Kapitel 12.1).

Etwas durchlaufen, das keine Liste ist:

kB_nicht_iterierbar.py

```
1 for i in 5:
2     print(i)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object isn't iterable
```

„isn't iterable" heißt: Man kann es nicht Element für Element durchlaufen. Eine Zahl ist keine Folge von Werten. Richtig: `for i in range(5):`.

`ValueError` – der Wert passt nicht

kB_wert.py

```
1 eingabe = input("Gib eine Zahl ein: ")
2 zahl = int(eingabe)
3 print("Das Doppelte:", zahl * 2)
```

Terminal

```
Gib eine Zahl ein: abc

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid syntax for integer with base 10: 'abc'
```

Der Typ stimmt (ein String), aber sein **Inhalt** lässt sich nicht in eine ganze Zahl umwandeln. Dasselbe passiert bei `int("3.5")` oder `float("3,5")` (Komma statt Punkt). Du kannst vorher prüfen, ob nur Ziffern eingetippt wurden:

```
eingabe = input("Gib eine Zahl ein: ")
if eingabe.isdigit():
    print("Das Doppelte:", int(eingabe) * 2)
else:
    print("Das war keine Zahl.")
```

`IndexError` – diese Stelle gibt es in der Liste nicht

kB_index.py

```
1 tage = ["Mo", "Di", "Mi", "Do", "Fr"]
2 for i in range(6):
3     print(tage[i])
```

Terminal

```
Mo
Di
Mi
Do
Fr

Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError: list index out of range
```

Die Liste hat 5 Elemente mit den Indizes 0 bis 4. `range(6)` zählt aber bis 5, und `tage[5]` gibt es nicht. Der größte Index ist immer `len(liste) - 1` (Kapitel 13.1). Am sichersten ist es, die Liste direkt zu durchlaufen: `for tag in tage:`. Dieselbe Meldung kommt bei `liste.pop()` auf eine leere Liste und bei Strings: `"Hallo"[10]`.

`ZeroDivisionError` – Teilen durch null

kB_null.py

```
noten = []
schnitt = sum(noten) / len(noten)
print("Durchschnitt:", schnitt)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: divide by zero
```

Die Liste ist leer, also ist `len(noten)` gleich 0. Durch 0 kann man nicht teilen, auch nicht mit `//` oder `%`. Prüfe vorher:

```
if len(noten) > 0:
    print("Durchschnitt:", sum(noten) / len(noten))
else:
    print("Noch keine Noten.")
```

`OSError` – Problem mit einer Datei

kB_datei.py

```
with open("wetterdaten.txt", "r") as datei:
    daten = datei.read()
print(daten)
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] ENOENT
```

Eine Datei, die es nicht gibt, kann man nicht zum Lesen öffnen. `[Errno 2]` ist ein Fehlercode, `ENOENT` steht für „no entry“: kein Eintrag gefunden. Typisch ist das beim ersten Start eines Programms, das einen Highscore liest (Kapitel 15). Im Emulator sind gespeicherte Dateien außerdem nach dem Neuladen der Seite weg. Prüfe vorher mit `os.listdir()`:

```
import os

if "wetterdaten.txt" in os.listdir():
    with open("wetterdaten.txt", "r") as datei:
        print(datei.read())
else:
    print("Noch keine Daten gespeichert.")
```

`RuntimeError` – eine Funktion ruft sich endlos selbst auf

kB_rekursion.py

```
1 def summe_bis(n):
2     # addiert n + (n-1) + ... – aber wo hoert das auf?
3     return n + summe_bis(n - 1)
4
5 print(summe_bis(10))
```

Die Meldung ist sehr lang, weil für jeden der vielen Aufrufe eine `File`-Zeile erscheint. Hier ihr Anfang und ihr Ende:

Terminal (gekürzt)

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "<stdin>", line 3, in summe_bis
  File "<stdin>", line 3, in summe_bis
  …
  File "<stdin>", line 3, in summe_bis
RuntimeError: maximum recursion depth exceeded
```

`summe_bis(10)` ruft `summe_bis(9)` auf, das `summe_bis(8)` und so weiter, über 0 hinaus ins Negative, ohne Ende. Irgendwann ist der Speicher für die Aufrufe voll („maximum recursion depth exceeded“: größte Aufruftiefe überschritten). Jede Funktion, die sich selbst aufruft, braucht eine **Abbruchbedingung** (Kapitel 16.6):

```
def summe_bis(n):
    if n == 0:
        return 0
    return n + summe_bis(n - 1)
```

`MemoryError` – der Speicher ist voll

kB_speicher.py

```
1 # hundert Millionen Messwerte auf einmal
2 messwerte = [0] * 100000000
3 print("geschafft:", len(messwerte))
```

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
MemoryError: memory allocation failed, allocating 400000000 bytes
```

Der PicoBoy hat nur wenige hundert Kilobyte Arbeitsspeicher. Eine Liste mit hundert Millionen Zahlen braucht rund 400 Millionen Bytes – das passt auf keinen Fall. Auf dem Gerät reicht schon eine Liste mit einer Million Zahlen für diese Meldung. Speicher verbrauchen vor allem sehr lange Listen, viele Bilder und Sprites und jeder Canvas (ein Canvas für den ganzen Bildschirm belegt rund

134 KB). Lege deshalb nur einen großen Canvas an (Kapitel 17) und lösche Elemente aus Listen, die du nicht mehr brauchst. Der Emulator hat nicht genau so viel Speicher wie das Gerät: Ein Programm, das im Emulator gerade noch läuft, kann auf dem Gerät einen `MemoryError` auslösen.

Die Hinweisfenster von PBC+ Studio

Nach einem Fehler öffnet PBC+ Studio ein Fenster mit einer Erklärung. Seine Überschrift verrät dir die Fehlerart auf Deutsch:

Fehlerart im Terminal	Überschrift des Hinweisfensters
`SyntaxError`	Schreibfehler im Code
`IndentationError`	Einrückung stimmt nicht
`NameError`	Unbekannter Name
`AttributeError`	Unbekannte Funktion oder Eigenschaft
`ImportError`	Modul nicht gefunden
`TypeError`	Falscher Typ
`ValueError`	Eingabe ist keine Zahl
`IndexError`	Listen-Position außerhalb
`ZeroDivisionError`	Division durch Null
`RuntimeError` (Rekursion)	Endlose Selbst-Aufrufe
`MemoryError`	Nicht genug Speicher
`OSError` (Datei fehlt)	Datei nicht gefunden

Der Text im Fenster passt nicht immer genau zu deinem Programm. Er nennt die häufigsten Ursachen. Bei einem `ValueError` lautet die Überschrift zum Beispiel immer „Eingabe ist keine Zahl“, auch wenn der Fehler gar nichts mit einer Eingabe zu tun hat. Verlass dich deshalb vor allem auf die Meldung im Terminal.

Wenn keine Fehlermeldung kommt, das Programm aber falsch ist

Die schwierigsten Fehler sind die ohne Meldung: Das Programm läuft, macht aber etwas anderes, als du willst. Dieses Programm soll 1 + 2 + … + 10 rechnen. Das Ergebnis müsste 55 sein:

kB_suche.py

```
1 # 1 + 2 + ... + 10 (soll 55 sein)
2 summe = 0
3 for zahl in range(1, 10):
4     summe = summe + zahl
5 print("Summe:", summe)
```

Terminal

```
Summe: 45
```

Python sieht hier keinen Fehler. Um herauszufinden, was passiert, lässt du dir mit `print` anzeigen, welche Werte die Variablen haben, während das Programm läuft:

kB_suche_print.py

```
# 1 + 2 + ... + 10 (soll 55 sein)
summe = 0
for zahl in range(1, 10):
    summe = summe + zahl
    print("zahl =", zahl, " summe =", summe)   # zur Fehlersuche
print("Summe:", summe)
```

Terminal

```
zahl = 1  summe = 1
zahl = 2  summe = 3
zahl = 3  summe = 6
zahl = 4  summe = 10
zahl = 5  summe = 15
zahl = 6  summe = 21
zahl = 7  summe = 28
zahl = 8  summe = 36
zahl = 9  summe = 45
Summe: 45
```

Jetzt sieht man es sofort: Die letzte Zahl ist 9, nicht 10. `range(1, 10)` hört **vor** der 10 auf. Richtig ist `range(1, 11)`. Ist der Fehler gefunden, löschst du die zusätzliche `print`-Zeile wieder (oder machst sie mit `#` zum Kommentar).

So gehst du vor (mehr dazu in Kapitel 22.3):

- Gib mit `print` die wichtigen Variablen aus, am besten mit Namen: `print("x =", x, "dx =", dx)`. Bei Spielen auf dem Display hilft die Konsole, weil der Bildschirm frei bleibt.
- Setze ein `print("bin hier")` in einen `if`-Zweig, um zu sehen, ob er überhaupt ausgeführt wird.
- Schnelle Schleifen erzeugen sehr viele Zeilen. Baue vorübergehend ein `time.sleep(0.5)` ein oder gib nur bei bestimmten Werten etwas aus.
- Verkleinere das Programm: Schalte Teile mit `#` ab, bis der Fehler verschwindet. Dann liegt er im zuletzt abgeschalteten Teil.
- Häufige Ursachen ohne Meldung: `range` endet eine Zahl zu früh, eine Zeile ist falsch eingerückt (sie gehört zur Schleife oder eben nicht), `=` statt `+=`, Koordinaten vertauscht, eine Bedingung mit `and` statt `or`.

Das Wichtigste in Kürze

- Fehlermeldungen von unten lesen: Fehlerart, Beschreibung, dann Zeilennummer.
- `SyntaxError` / `IndentationError`: Das Programm startet gar nicht. Fehler in der genannten Zeile oder kurz davor.
- `NameError`, `AttributeError`: Tippfehler im Namen, fehlender `import`.
- `TypeError`: Text und Zahl gemischt, falsche Anzahl Argumente, `return` vergessen.
- `ValueError`, `IndexError`, `ZeroDivisionError`, `OSError`: Ein Wert, eine Stelle oder eine Datei passt nicht. Vorher mit `if` prüfen.
- Ohne Fehlermeldung: mit `print` nachsehen, welche Werte die Variablen wirklich haben.

Lösungen zu Kapitel 1: Der PicoBoy Color Plus und PBC+ Studio

Aufgabe 1 – Teile finden

Der Joystick-Druck (MITTE) ist im Emulator die **Leertaste**. Ein Programm hältst du mit ■ **Stop** an. Fehlermeldungen erscheinen im **Terminal** unten in PBC+ Studio (und zusätzlich in einem Hinweisfenster).

Aufgabe 2 – Dein Name (`k01_loesung2.py`)

k01_loesung2.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Hallo, Mia!", 40, 120, pbc.YELLOW)
```



Der Text ist länger als „Bereit?“. Deshalb beginnt er weiter links (x = 40), sonst würde er rechts abgeschnitten.

Aufgabe 3 – Speichern und wieder öffnen

Nach **Dateien → Vom Computer…** steht das Programm genau so im Editor, wie du es gespeichert hast. Die Datei `hallo.py` liegt im Download-Ordner (oder in dem Ordner, den du beim Speichern gewählt hast).

Aufgabe 4 – Die Hilfe benutzen

Das Beispiel fügt drei Zeilen mit `pbc.text12`, `pbc.text18` und `pbc.text24` ein. Die größte Schrift ist `text24`.

Aufgabe 5 – Auf das echte Gerät

Nach dem Einschalten zeigt das Startmenü den Dateinamen, unter dem du das Programm gespeichert hast. Wähle ihn mit dem Joystick aus und starte ihn.

Lösungen zu Kapitel 2: Das erste Programm

Aufgabe 1 – Wo erscheint was?

„oben“ steht weiß in der linken oberen Ecke (x = 20, y = 20), „Mitte“ grün ungefähr in der Mitte (x = 95, y = 130), „unten“ rot rechts unten (x = 150, y = 240). Je größer x, desto weiter rechts; je größer y, desto weiter unten.



Aufgabe 2 – Drei Zeilen über dich (`k02_loesung2.py`)

k02_loesung2.py

```
1 import pbc
2
3 pbc.clear()
4 pbc.text24("Jonas", 20, 40, pbc.YELLOW)
5 pbc.text18("Klasse 8a", 20, 90, pbc.WHITE)
6 pbc.text18("Hobby: Fussball", 20, 120, pbc.CYAN)
```



Die Zeilen mit `text18` haben 30 Pixel Abstand, damit sie sich nicht berühren.

Aufgabe 3 – Finde die Fehler (`k02_loesung3.py`)

Python meldet die Fehler **einzeln**, nacheinander:

1. `SyntaxError` in Zeile 5: Hinter `14 Uhr` fehlt das schließende Anführungszeichen. Python hält deshalb alles bis zum Zeilenende für Text.
2. Nach der Korrektur: `AttributeError` in Zeile 4: `pbc.Text24` gibt es nicht, richtig ist `pbc.text24` (klein).
3. Danach: `AttributeError` in Zeile 6: `pbc.white` gibt es nicht, richtig ist `pbc.WHITE`.

k02_loesung3.py

```
import pbc

pbc.clear()
pbc.text24("Sommerfest", 30, 60, pbc.YELLOW)
pbc.text18("Freitag, 14 Uhr", 50, 120, pbc.WHITE)
pbc.text18("Schulhof", 80, 150, pbc.WHITE)
```



Außerdem wurden die x-Werte angepasst, damit die Texte ungefähr mittig stehen. Das war kein Fehler, sieht aber besser aus.

Aufgabe 4 – Ein Schild (`k02_loesung4.py`)

k02_loesung4.py

```
import pbc

pbc.clear(pbc.RED)
pbc.text24("STOP", 85, 110, pbc.WHITE)
pbc.text12("Bitte warten", 85, 160, pbc.WHITE)
```



`pbc.clear(pbc.RED)` füllt den ganzen Bildschirm rot. Die Texte müssen **danach** kommen, sonst würden sie übermalt.

Aufgabe 5 – Stundenplan (`k02_loesung5.py`)

k02_loesung5.py

```
# Stundenplan fuer Montag in der 8x8-Schrift: jedes Zeichen ist
# genau 8 Pixel breit, so stehen die Spalten sauber untereinander.
import pbc

pbc.clear()
pbc.text18("Montag", 20, 20, pbc.YELLOW)
pbc.text("Std  Fach      Raum", 20, 60, pbc.CYAN)
pbc.text("1    Mathe     104", 20, 80, pbc.WHITE)
pbc.text("2    Mathe     104", 20, 95, pbc.WHITE)
pbc.text("3    Englisch  211", 20, 110, pbc.WHITE)
pbc.text("4    Englisch  211", 20, 125, pbc.WHITE)
pbc.text("5    Sport     Halle", 20, 140, pbc.WHITE)
pbc.text("6    Sport     Halle", 20, 155, pbc.WHITE)
```

Jedes Zeichen von `pbc.text` ist 8 Pixel breit, auch ein Leerzeichen. Stehen in jeder Zeile gleich viele Zeichen vor einer Spalte, beginnt die Spalte überall an derselben x-Position. Die Zeichen sind 8 Pixel hoch; der Zeilenabstand muss also mindestens 8 Pixel sein. Mit 15 Pixeln ist die Tabelle gut lesbar.



Lösungen zu Kapitel 3: Die Konsole: `print` und `input`

Aufgabe 1 – Was steht im Terminal?

```
Terminal
Eins Zwei
DreiVier

Fünf 6 Sieben8
12 1 2
```

- Zwischen Werten, die durch **Kommas** getrennt sind, setzt `print` ein Leerzeichen: „Eins Zwei“.
- `+` klebt Texte **ohne** Leerzeichen aneinander: „DreiVier“, „Sieben8“.
- `print()` gibt eine leere Zeile aus.
- `"1" + "2"` sind zwei Texte, also entsteht „12“ und nicht 3.

Aufgabe 2 – Ein Steckbrief (`k03_loesung2.py`)

k03_loesung2.py

```python
# Steckbrief in der Konsole
print("==========================")
print("  Steckbrief: Erdmännchen")
print("==========================")
print("Lebensraum:", "Wüsten in Afrika")
print("Größe:", 30, "cm")
print("Gewicht:", 0.8, "kg")
print("Futter:", "Insekten, Eier, Früchte")
print()
print("Besonderheit: Einer hält immer Wache!")
```

```
Terminal
==========================
  Steckbrief: Erdmännchen
==========================
Lebensraum: Wüsten in Afrika
Größe: 30 cm
Gewicht: 0.8 kg
Futter: Insekten, Eier, Früchte

Besonderheit: Einer hält immer Wache!
```

Die Zahlen `30` und `0.8` stehen ohne Anführungszeichen. `print` schreibt sie so hin, wie sie sind. In der Konsole sind Umlaute erlaubt.

Aufgabe 3 – Finde den Fehler (`k03_loesung3.py`)

Das fehlerhafte Programm meldet:

Terminal (Eingabe: 2026)

```
Welches Jahr haben wir? 2026

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported types for __add__: 'str', 'int'
```

`input` liefert den Text `"2026"`. In Zeile 2 soll zu diesem Text die Zahl 1 addiert werden, das geht nicht (`TypeError`). Die Antwort muss mit `int` in eine Zahl umgewandelt werden:

k03_loesung3.py

```
1 jahr = int(input("Welches Jahr haben wir? "))
2 print("Nächstes Jahr ist", jahr + 1)
```

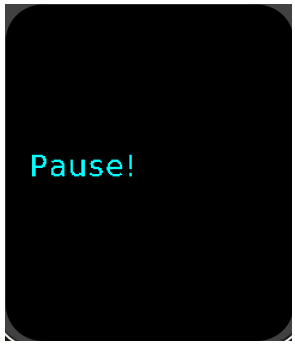
Terminal (Eingabe: 2026)

```
Welches Jahr haben wir? 2026
Nächstes Jahr ist 2027
```

Aufgabe 4 – Ein Schild zum Eintippen (`k03_loesung4.py`)

k03_loesung4.py

```
1 import pbc
2
3 wort = input("Was soll auf dem Display stehen? ")
4
5 pbc.clear()
6 pbc.text24(wort, 20, 120, pbc.CYAN)
7 print("Auf dem Display steht jetzt:", wort)
```



Terminal (Eingabe: Pause!)

```
Was soll auf dem Display stehen? Pause!
Auf dem Display steht jetzt: Pause!
```

Die Antwort ist ein Text, deshalb kann sie direkt an `pbc.text24` gehen.

Aufgabe 5 – Die Kinokarte (`k03_loesung5.py`)

k03_loesung5.py

```
1 # Eine Kinokarte
2 import pbc
3
4 film = input("Welcher Film? ")
```

```
 5 uhrzeit = input("Um wie viel Uhr? ")
 6 saal = int(input("In welchem Saal? "))
 7
 8 print("Deine Karte:", film, "um", uhrzeit, "in Saal", saal)
 9
10 pbc.clear(pbc.WHITE)
11 pbc.text18("KINOKARTE", 20, 40, pbc.RED)
12 pbc.text24(film, 20, 90, pbc.BLACK)
13 pbc.text18("Beginn: " + uhrzeit + " Uhr", 20, 140, pbc.BLACK)
14 pbc.text18("Saal " + str(saal), 20, 170, pbc.BLACK)
15 pbc.text12("Viel Spass!", 20, 230, pbc.BLUE)
```

Terminal (Eingaben: Dino-Park, 17:30, 4)

```
Welcher Film? Dino-Park
Um wie viel Uhr? 17:30
In welchem Saal? 4
Deine Karte: Dino-Park um 17:30 in Saal 4
```

Die Uhrzeit bleibt ein Text, denn `17:30` ist keine Zahl, mit der man rechnen könnte. Der Saal wird mit `int` zur Zahl und muss für das Display in Zeile 14 mit `str` wieder zum Text werden, damit `+` funktioniert.



Aufgabe 6 – Text oder Zahl? (`k03_loesung6.py`)

k03_loesung6.py

```
1 print("3" + "4")
2 print(3 + 4)
3 print(int("3") + int("4"))
4 print(str(3) + str(4))
5 print(float("3") + 4)
```

Terminal

```
34
7
7
34
7.0
```

Zwei Texte werden mit `+` aneinandergehängt (`34`), zwei Zahlen addiert (`7`). `int` und `str` wandeln vorher um. Kommazahl plus ganze Zahl ergibt eine Kommazahl (`7.0`). Die REPL zeigt Texte mit Anführungszeichen an: `'34'`.

Lösungen zu Kapitel 4: Variablen und Rechnen

Aufgabe 1 – Was gibt das Programm aus?

Terminal

```
8 14
3 2 8
3.0 abab
```

- Zeile 2 rechnet mit dem **damaligen** Wert von `x`: `y` wird 14. Dass `x` in Zeile 3 danach 8 wird, ändert `y` nicht mehr.
- `17 // 5` ist 3, `17 % 5` ist 2 (17 = 3 · 5 + 2), `2 ** 3` ist 8.
- `9 / 3` ergibt `3.0`: `/` liefert immer eine Kommazahl.
- `"ab" * 2` wiederholt den Text: `abab`.

Aufgabe 2 – Finde den Fehler (`k04_loesung2.py`)

Das fehlerhafte Programm gibt Folgendes aus:

Terminal (Eingabe: 1.5)

```
Preis pro Heft in Euro: 1.5
4 Hefte kosten 1.51.51.51.5 Euro.
```

`input` liefert den **Text** `"1.5"`. Ein Text mal 4 ist erlaubt, er wird dann viermal hintereinander geschrieben. Deshalb gibt es keine Fehlermeldung. Mit `float` wird die Eingabe zur Zahl, und `:.2f` sorgt für zwei Nachkommastellen:

k04_loesung2.py

```
1 preis = float(input("Preis pro Heft in Euro: "))
2 anzahl = 4
3 gesamt = preis * anzahl
4 print(f"{anzahl} Hefte kosten {gesamt:.2f} Euro.")
```

Terminal (Eingabe: 1.5)

```
Preis pro Heft in Euro: 1.5
4 Hefte kosten 6.00 Euro.
```

Aufgabe 3 – Sekunden umrechnen (`k04_loesung3.py`)

k04_loesung3.py

```
1 sekunden = int(input("Wie viele Sekunden? "))
2
3 stunden = sekunden // 3600
4 rest = sekunden % 3600
5 minuten = rest // 60
6 rest_sekunden = rest % 60
7
8 print(f"{sekunden} s = {stunden} h {minuten} min {rest_sekunden} s")
```

Terminal (Eingabe: 5000)

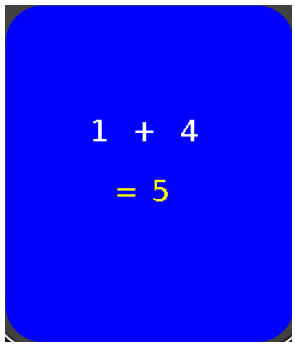
```
Wie viele Sekunden? 5000
5000 s = 1 h 23 min 20 s
```

`sekunden // 3600` sind die ganzen Stunden, `sekunden % 3600` die Sekunden, die danach noch übrig sind. Aus diesem Rest holt man genauso die Minuten (`// 60`) und die restlichen Sekunden (`% 60`).

Aufgabe 4 – Zwei Würfel (`k04_loesung4.py`)

k04_loesung4.py

```
1 import pbc
2 import random
3
4 wurf1 = random.randint(1, 6)
5 wurf2 = random.randint(1, 6)
6 summe = wurf1 + wurf2
7
8 pbc.clear(pbc.BLUE)
9 pbc.text24(f"{wurf1}  +  {wurf2}", 70, 90, pbc.WHITE)
10 pbc.text24(f"= {summe}", 90, 140, pbc.YELLOW)
11 print(f"Gewürfelt: {wurf1} und {wurf2}, zusammen {summe}")
```



Terminal (Beispiel)

```
Gewürfelt: 1 und 4, zusammen 5
```

Jeder Würfel braucht seinen eigenen Aufruf von `random.randint`. Die f-Strings setzen die Zahlen direkt in den Text für das Display ein, `str()` ist nicht nötig.

Aufgabe 5 – Pizza-Party (`k04_loesung5.py`)

k04_loesung5.py

```
1 # Pizza-Party
2 STUECKE_PRO_PIZZA = 8
3
4 personen = int(input("Wie viele Personen kommen? "))
```

```python
hunger = int(input("Wie viele Stücke isst jede Person? "))

stuecke = personen * hunger
# Aufrunden: 7 dazuzählen, dann ganzzahlig teilen
pizzen = (stuecke + STUECKE_PRO_PIZZA - 1) // STUECKE_PRO_PIZZA
uebrig = pizzen * STUECKE_PRO_PIZZA - stuecke

print(f"Ihr braucht {stuecke} Stücke.")
print(f"Bestellt {pizzen} Pizzen.")
print(f"Dann bleiben {uebrig} Stücke übrig.")
```

Terminal (Eingaben: 7 und 3)

```
Wie viele Personen kommen? 7
Wie viele Stücke isst jede Person? 3
Ihr braucht 21 Stücke.
Bestellt 3 Pizzen.
Dann bleiben 3 Stücke übrig.
```

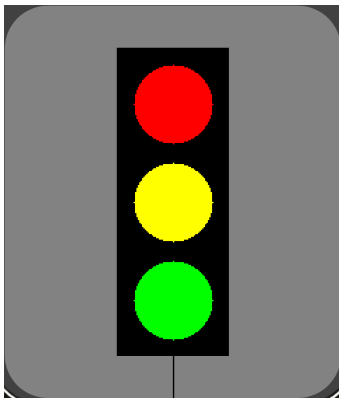
`stuecke // 8` würde **abrunden**: 21 // 8 ergibt 2, zwei Pizzen reichen aber nicht. Zählt man vorher 7 dazu, landet man bei jeder angebrochenen Pizza über der nächsten Achtergrenze: (21 + 7) // 8 = 28 // 8 = 3. Geht die Rechnung genau auf, z. B. bei 24 Stücken, ändert das nichts: (24 + 7) // 8 = 3. Die Anzahl der Stücke pro Pizza steht als Konstante `STUECKE_PRO_PIZZA` am Anfang. Großgeschriebene Namen zeigen: Dieser Wert wird im Programm nicht verändert.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 5: Zeichnen und Farben

Aufgabe 1 – Was wird gezeichnet?

Eine **Ampel**: Der Hintergrund ist grau. Ein schwarzes Rechteck (80 Pixel breit, 220 hoch) bildet das Gehäuse. Darin liegen drei gefüllte Kreise mit Radius 28 genau übereinander, weil alle den Mittelpunkt x = 120 haben: oben rot, in der Mitte gelb, unten grün. Die senkrechte Linie unten ist der Mast. Die Kreise sind erst **nach** dem Gehäuse gezeichnet, sonst wären sie verdeckt.



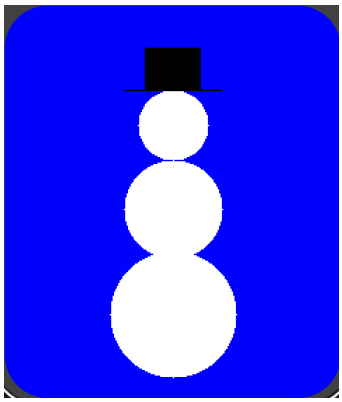
Aufgabe 2 – Finde die Fehler (`k05_loesung2.py`)

1. `SyntaxError` in Zeile 7: Vor `fill=True` fehlt das Komma.
2. Nach der Korrektur: `TypeError` in Zeile 10: `pbc.hline` fehlt die Farbe.
3. Ohne Fehlermeldung: Der Hut sitzt rechts neben dem Kopf. Bei `fill_rect` ist (x, y) die **linke obere Ecke**, nicht die Mitte. Damit der 40 Pixel breite Hut mittig über x = 120 sitzt, muss er bei 120 − 20 = 100 beginnen.

k05_loesung2.py

```python
import pbc

pbc.clear(pbc.BLUE)
# Drei Kugeln uebereinander, in der Mitte
pbc.ellipse(120, 220, 45, 45, pbc.WHITE, fill=True)
pbc.ellipse(120, 145, 35, 35, pbc.WHITE, fill=True)
pbc.ellipse(120, 85, 25, 25, pbc.WHITE, fill=True)
# Hut: 40 breit, mittig auf dem Kopf
pbc.fill_rect(100, 30, 40, 30, pbc.BLACK)
pbc.hline(85, 60, 70, pbc.BLACK)
```



Aufgabe 3 – Eine Flagge (`k05_loesung3.py`)

k05_loesung3.py

```python
import pbc

gold = pbc.rgb(255, 204, 0)
streifen = pbc.HEIGHT // 3
print("Streifenhoehe:", streifen)

pbc.fill_rect(0, 0, pbc.WIDTH, streifen, pbc.BLACK)
pbc.fill_rect(0, streifen, pbc.WIDTH, streifen, pbc.RED)
# Der letzte Streifen reicht bis ganz unten
pbc.fill_rect(0, 2 * streifen, pbc.WIDTH, pbc.HEIGHT - 2 * streifen, gold)
```

Terminal

```
Streifenhoehe: 93
```

280 lässt sich nicht ohne Rest durch 3 teilen: `280 // 3` ergibt 93, drei Streifen wären zusammen nur 279 Pixel hoch. Die letzte Pixelzeile bliebe schwarz. Deshalb ist der letzte Streifen `pbc.HEIGHT - 2 * streifen` = 94 Pixel hoch und reicht genau bis unten.



Aufgabe 4 – Die Zielscheibe (`k05_loesung4.py`)

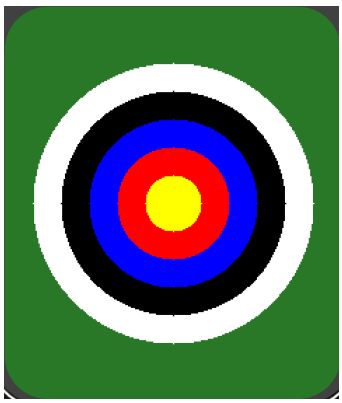
k05_loesung4.py

```
import pbc

mitte_x = pbc.WIDTH // 2
mitte_y = pbc.HEIGHT // 2

pbc.clear(pbc.rgb(40, 120, 40))
# Von aussen nach innen: das Groesste zuerst
pbc.ellipse(mitte_x, mitte_y, 100, 100, pbc.WHITE, fill=True)
pbc.ellipse(mitte_x, mitte_y, 80, 80, pbc.BLACK, fill=True)
pbc.ellipse(mitte_x, mitte_y, 60, 60, pbc.BLUE, fill=True)
pbc.ellipse(mitte_x, mitte_y, 40, 40, pbc.RED, fill=True)
pbc.ellipse(mitte_x, mitte_y, 20, 20, pbc.YELLOW, fill=True)
```

Die Kreise müssen **vom größten zum kleinsten** gezeichnet werden. Jeder gefüllte Kreis deckt alles unter sich zu. Würde man den kleinen gelben Kreis zuerst zeichnen, wäre er unter den größeren verschwunden. Von jedem Kreis bleibt nur ein 20 Pixel breiter Ring sichtbar, weil der nächste Radius um 20 kleiner ist.



Aufgabe 5 – Ein Balkendiagramm (`k05_loesung5.py`)

k05_loesung5.py

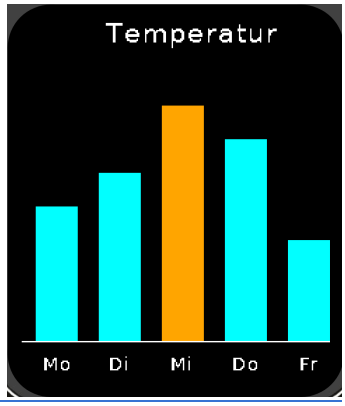
```
import pbc

# Hoechsttemperaturen einer Woche in Grad
mo = 12
di = 15
mi = 21
do = 18
fr = 9
FAKTOR = 8      # 1 Grad = 8 Pixel hoch
BODEN = 240     # y der Grundlinie

pbc.clear()
pbc.text18("Temperatur", 70, 10, pbc.WHITE)
pbc.hline(10, BODEN, 220, pbc.WHITE)
# Balken: oben = BODEN - Hoehe, weil y nach unten waechst
pbc.fill_rect(20, BODEN - mo * FAKTOR, 30, mo * FAKTOR, pbc.CYAN)
pbc.fill_rect(65, BODEN - di * FAKTOR, 30, di * FAKTOR, pbc.CYAN)
pbc.fill_rect(110, BODEN - mi * FAKTOR, 30, mi * FAKTOR, pbc.ORANGE)
pbc.fill_rect(155, BODEN - do * FAKTOR, 30, do * FAKTOR, pbc.CYAN)
pbc.fill_rect(200, BODEN - fr * FAKTOR, 30, fr * FAKTOR, pbc.CYAN)
```

```
# Beschriftung unter den Balken
pbc.text12("Mo", 25, 248, pbc.WHITE)
pbc.text12("Di", 72, 248, pbc.WHITE)
pbc.text12("Mi", 116, 248, pbc.WHITE)
pbc.text12("Do", 160, 248, pbc.WHITE)
pbc.text12("Fr", 208, 248, pbc.WHITE)
```

Ein Balken mit der Höhe `mi * FAKTOR` = 168 Pixel beginnt oben bei `240 - 168` = 72 und reicht bis 240. Weil x und y mit Variablen berechnet werden, passt das Diagramm sich an, wenn du andere Temperaturen einträgst. Der wärmste Tag ist von Hand orange gefärbt. Automatisch den größten Wert finden kannst du erst mit Entscheidungen (Kapitel 7) oder mit `max` (Kapitel 13). Viele fast gleiche Zeilen schreien nach einer Schleife: Die lernst du im nächsten Kapitel.



ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 6: Schleifen und Zeit

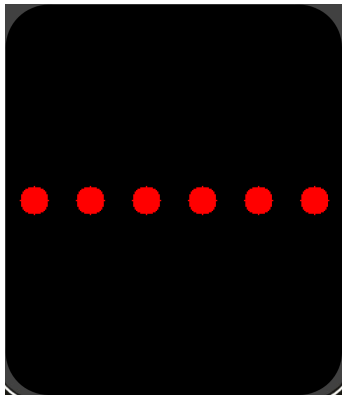
Aufgabe 1 – Was gibt das Programm aus?

```
Terminal
2
5
8
11
---
k ist 3
k ist 2
k ist 1
doppelt 2
```

- `range(2, 12, 3)` beginnt bei 2 und geht in Dreierschritten weiter: 2, 5, 8, 11. Die nächste Zahl wäre 14, sie ist größer als 12.
- `range(3, 0, -1)` zählt rückwärts 3, 2, 1. Die 0 ist nicht dabei.
- Die letzte Zeile ist **nicht** eingerückt. Sie läuft nur einmal, nach der Schleife. Die Schleifenvariable `k` behält ihren letzten Wert 1, deshalb steht da „doppelt 2“.

Aufgabe 2 – Finde die Fehler (`k06_loesung2.py`)

1. `SyntaxError` in Zeile 5: Am Ende des Schleifenkopfs fehlt der Doppelpunkt.
2. Ohne Fehlermeldung: `range(1, 6)` liefert nur 1 bis 5, also fünf Punkte, und der erste liegt bei `20 + 1 * 40` = 60. Mit `range(6)` ist `i` 0 bis 5: sechs Punkte von x = 20 bis x = 220.



k06_loesung2.py

```
import pbc

pbc.clear()
# Sechs rote Punkte nebeneinander, von x = 20 bis x = 220
for i in range(6):
    x = 20 + i * 40
    pbc.ellipse(x, 140, 10, 10, pbc.RED, fill=True)
```

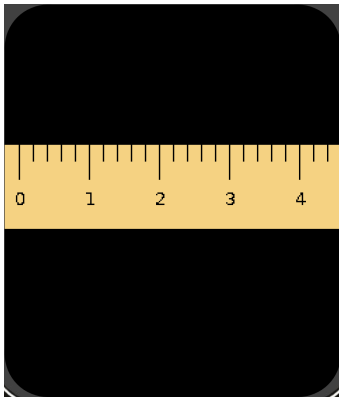
Aufgabe 3 – Ein Lineal (`k06_loesung3.py`)

k06_loesung3.py

```
import pbc

holz = pbc.rgb(240, 210, 130)
pbc.clear()
pbc.fill_rect(0, 100, pbc.WIDTH, 60, holz)
# Kurze Striche alle 10 Pixel
for x in range(10, 231, 10):
    pbc.vline(x, 100, 12, pbc.BLACK)
# Lange Striche alle 50 Pixel, darunter die Zahl
for x in range(10, 231, 50):
    pbc.vline(x, 100, 25, pbc.BLACK)
    pbc.text12(f"{(x - 10) // 50}", x - 3, 130, pbc.BLACK)
```

Die erste Schleife zeichnet 23 kurze Striche (x = 10, 20 … 230). Die zweite zeichnet an jeder fünften Stelle einen längeren Strich darüber. Die Zahl darunter berechnet Zeile 12 aus `x`: Bei x = 10 ist `(10 - 10) // 50` = 0, bei x = 60 ist sie 1 und so weiter. Ein Strich ist 1 Pixel breit; die Zahl beginnt 3 Pixel weiter links, damit sie ungefähr mittig darunter steht.



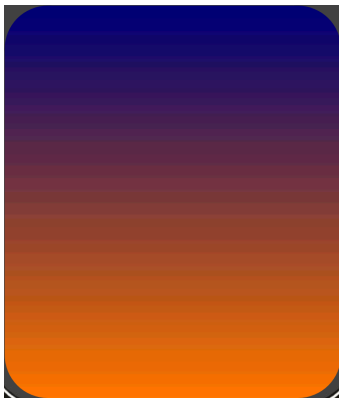
Aufgabe 4 – Abendhimmel (`k06_loesung4.py`)

k06_loesung4.py

```
import pbc

# Jede Pixelzeile bekommt ihre eigene Farbe
for y in range(pbc.HEIGHT):
    rot = y * 255 // 279
    gruen = y * 120 // 279
    blau = 120 - y * 120 // 279
    pbc.hline(0, y, pbc.WIDTH, pbc.rgb(rot, gruen, blau))
```

Die Schleife läuft **280-mal**, einmal für jede Pixelzeile (y = 0 bis 279). Oben (y = 0) ist die Farbe `rgb(0, 0, 120)`, ein dunkles Blau. Unten (y = 279) ist sie `rgb(255, 120, 0)`, ein Orange. Dazwischen wächst der Rot- und Grünanteil mit `y`, der Blauanteil nimmt ab. Erst multiplizieren, dann mit `//` teilen: `y * 255 // 279` bleibt eine ganze Zahl und wird bei y = 279 genau 255.



Aufgabe 5 – Laufschrift (`k06_loesung5.py`)

k06_loesung5.py

```
import pbc
import time

text = "Heute Sportfest!"
x = 0
pbc.clear()
while True:
    pbc.text18(text, x, 130, pbc.YELLOW)
    time.sleep_ms(30)
    # Derselbe Text in Schwarz loescht ihn
    pbc.text18(text, x, 130, pbc.BLACK)
    x = (x + 3) % pbc.WIDTH
```



Wie beim Ball in Abschnitt 6.6: zeichnen, warten, an derselben Stelle in der Hintergrundfarbe übermalen, weiterrücken. Weil der Text mit `pbc.text18(…, pbc.BLACK)` genau dieselben Pixel trifft, verschwindet er vollständig. `x = (x + 3) % pbc.WIDTH` sorgt dafür, dass der Text nach dem rechten Rand wieder links beginnt. Rechts wird er dabei abgeschnitten, und er springt, sobald `x` wieder 0 wird. Einen Text, der weich hinausläuft und von der anderen Seite hereinkommt, baust du später mit Entscheidungen (Kapitel 7).

Lösungen zu Kapitel 7: Entscheidungen

Aufgabe 1 – Was gibt das Programm aus?

Terminal

```
B
D
E
G
```

- `punkte > 100` ist falsch, `punkte > 50` ist wahr: „B“. Das `else` wird übersprungen.
- `leben > 0` ist wahr, `not bonus` ist auch wahr (`bonus` ist `False`). Beide wahr, also ist die `and`-Bedingung wahr: „D“.
- `punkte < 50` ist falsch, aber `leben == 2` ist wahr. Für `or` reicht eins: „E“.
- `bonus` ist `False`, „F“ erscheint nicht.
- `print("G")` ist nicht eingerückt und läuft immer.

Aufgabe 2 – Finde die Fehler (`k07_loesung2.py`)

Python meldet zuerst einen `SyntaxError` in Zeile 3. Die vier Fehler:

1. Zeile 3: `=` statt `==`. Für „mindestens 18“ ist aber `>=` richtig (der Denkfehler: mit `==` wäre nur genau 18 volljährig).
2. Zeile 3: Am Ende fehlt der Doppelpunkt.
3. Zeile 6: Die Zeile unter `else:` muss eingerückt sein.
4. Zeile 1: `input` liefert Text. Ohne `int()` bricht der Vergleich `>=` mit einem `TypeError` ab.

k07_loesung2.py

```
1 alter = int(input("Wie alt bist du? "))
2
3 if alter >= 18:
4     print("Du bist volljährig.")
5 else:
6     print("Du bist noch nicht volljährig.")
```

Terminal

```
Wie alt bist du? 16
Du bist noch nicht volljährig.
```

Aufgabe 3 – Die Ergebnis-Ampel (`k07_loesung3.py`)

k07_loesung3.py

```
import pbc

punkte = int(input("Wie viele Punkte (0 bis 100)? "))

if punkte >= 80:
    farbe = pbc.GREEN
    meldung = "Super!"
elif punkte >= 50:
    farbe = pbc.YELLOW
    meldung = "Bestanden"
else:
    farbe = pbc.RED
    meldung = "Weiter ueben"

pbc.clear()
pbc.ellipse(120, 115, 70, 70, farbe, fill=True)
pbc.text24(f"{punkte} Punkte", 60, 205, pbc.WHITE)
pbc.text18(meldung, 70, 240, farbe)
```



Die Bedingungen werden von der höchsten Grenze aus geprüft: Bei 73 Punkten ist `punkte >= 80` falsch, `punkte >= 50` wahr, der Kreis wird gelb. In jedem Zweig werden nur die Variablen `farbe` und `meldung` belegt. Gezeichnet wird danach einmal für alle Fälle. So steht der Zeichencode nicht dreimal im Programm.

Aufgabe 4 – Schaltjahr (`k07_loesung4.py`)

k07_loesung4.py

```
jahr = int(input("Jahr: "))

if (jahr % 4 == 0 and jahr % 100 != 0) or jahr % 400 == 0:
    print(f"{jahr} ist ein Schaltjahr.")
else:
    print(f"{jahr} ist kein Schaltjahr.")
```

Terminal

```
Jahr: 1900
1900 ist kein Schaltjahr.
```

Die Klammer fasst die erste Regel zusammen: durch 4 teilbar **und** nicht durch 100. Das `or` fügt die Ausnahme hinzu. 2024 ist ein Schaltjahr, 2023 nicht, 1900 nicht (durch 100 teilbar), 2000 schon (durch 400 teilbar).

Aufgabe 5 – Papier falten (`k07_loesung5.py`)

k07_loesung5.py

```
dicke = 100            # in Mikrometern: 0,1 mm
everest = 8848000000   # 8848 m in Mikrometern
faltungen = 0

while dicke <= everest:
    dicke = dicke * 2
    faltungen += 1

print(f"Nach {faltungen} Faltungen ist der Stapel {dicke // 1000000} m dick.")
```

Terminal

```
Nach 27 Faltungen ist der Stapel 13421 m dick.
```

Die Schleife verdoppelt die Dicke, solange sie noch nicht größer als der Mount Everest ist. Wie oft das nötig ist, weiß man vorher nicht. Deshalb ist `while` hier die richtige Schleife. Nach 26 Faltungen sind es erst etwa 6700 m, nach 27 schon über 13 km. `dicke // 1000000` rechnet Mikrometer in ganze Meter um.

Aufgabe 6 – Kopfrechentrainer (`k07_loesung6.py`)

k07_loesung6.py

```
import random

richtig = 0
gestellt = 0

for runde in range(5):
    a = random.randint(2, 9)
    b = random.randint(2, 9)
    antwort = int(input(f"{a} mal {b} = "))
    if antwort == 0:
        break
    gestellt += 1
    if antwort == a * b:
        print("Richtig!")
        richtig += 1
    else:
        print(f"Falsch, richtig ist {a * b}.")

print(f"Du hast {richtig} von {gestellt} Aufgaben gelöst.")
if gestellt == 5 and richtig == 5:
    print("Perfekt!")
elif richtig >= 3:
    print("Gut gemacht!")
else:
    print("Übe noch ein bisschen.")
```

Hier ein Lauf, bei dem zweimal falsch geantwortet und dann 0 eingegeben wurde (die Aufgaben sind bei jedem Lauf anders):

```
Terminal
4 mal 7 = 1
Falsch, richtig ist 28.
8 mal 2 = 1
Falsch, richtig ist 16.
7 mal 6 = 0
Du hast 0 von 2 Aufgaben gelöst.
Übe noch ein bisschen.
```

- Die `for`-Schleife stellt höchstens fünf Aufgaben. `break` in Zeile 11 beendet sie vorzeitig, wenn 0 eingegeben wird.
- `gestellt` zählt die Aufgaben, die wirklich beantwortet wurden. Die 0 zählt nicht mit, weil Zeile 12 nach dem `break` nicht mehr drankommt.
- Das Lob am Ende prüft mit `if` / `elif` / `else` von oben nach unten. „Perfekt!“ gibt es nur, wenn alle fünf Aufgaben gestellt und richtig gelöst wurden.

Lösungen zu Kapitel 8: Tasten und Joystick

Aufgabe 1 – Was erscheint im Terminal?

```
Terminal
11
12
7
8
Ende bei 8
```

Jeder Druck zählt genau einmal, weil `was_pressed_…()` benutzt wird: hoch → 11, hoch → 12, runter → 12 − 5 = 7, hoch → 8. Der Druck auf A gibt „Ende bei 8“ aus und beendet die Schleife mit `break`.

Aufgabe 2 – Finde den Fehler (`k08_loesung2.py`)

In Zeile 9 fehlen die Klammern: `if pbc.was_pressed_b:` fragt keine Taste ab. `pbc.was_pressed_b` ohne Klammern ist die Funktion selbst, und die zählt als wahr. Deshalb zählt das Programm alle 100 ms einen Treffer:

```
Terminal (ohne Drücken)
Treffer: 1
Treffer: 2
Treffer: 3
Treffer: 4
```

```
Treffer: 5
Treffer: 6
Treffer: 7
Treffer: 8
Treffer: 9
Treffer: 10
Treffer: 11
Treffer: 12
Treffer: 13
```

Mit Klammern wird die Funktion aufgerufen, und es zählt nur jeder Druck auf B (hier zwei Drücke):

k08_loesung2.py

```python
import pbc
import time

treffer = 0
pbc.clear()
pbc.text18("Druecke B!", 75, 120, pbc.WHITE)

while True:
    if pbc.was_pressed_b():
        treffer += 1
        print("Treffer:", treffer)
    time.sleep_ms(100)
```

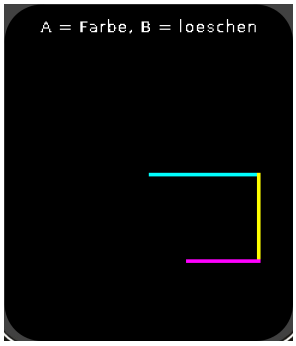
Terminal

```
Treffer: 1
Treffer: 2
```

Aufgabe 3 – Bessere Zaubertafel (`k08_loesung3.py`)

k08_loesung3.py

```
 1 import pbc
 2 import time
 3
 4 x = 120
 5 y = 140
 6 farbe = pbc.CYAN
 7
 8 pbc.clear()
 9 pbc.text12("A = Farbe, B = loeschen", 30, 10, pbc.WHITE)
10
11 while True:
12     # Am Rand bleibt der Stift stehen
13     if pbc.pressed_left() and x > 0:
14         x -= 1
15     if pbc.pressed_right() and x < 237:
16         x += 1
17     if pbc.pressed_up() and y > 30:
18         y -= 1
19     if pbc.pressed_down() and y < 277:
20         y += 1
21     if pbc.was_pressed_a():
22         if farbe == pbc.CYAN:
23             farbe = pbc.YELLOW
24         elif farbe == pbc.YELLOW:
25             farbe = pbc.MAGENTA
26         else:
27             farbe = pbc.CYAN
28     if pbc.pressed_b():
29         pbc.clear()
30     pbc.fill_rect(x, y, 3, 3, farbe)
31     time.sleep_ms(10)
```



- **Rand:** Jede Richtung wird nur ausgeführt, wenn der Stift noch Platz hat (`and`). Das Quadrat ist 3 Pixel groß, also darf `x` höchstens 237 und `y` höchstens 277 werden. Nach oben ist bei y = 30 Schluss, damit der Text oben frei bleibt.
- **Farbe:** Die Stiftfarbe steht in der Variablen `farbe`. Bei jedem Druck auf A (Zeile 21) prüft `if` / `elif` / `else`, welche Farbe gerade aktiv ist, und wählt die nächste. Hier muss `was_pressed_a()` stehen: Mit `pressed_a()` würde die Farbe bei jedem Durchlauf wechseln, solange A unten ist.
- Im Bild wurde nach rechts (cyan), nach unten (gelb) und nach links (magenta) gemalt.

Aufgabe 4 – Lichtschalter (`k08_loesung4.py`)

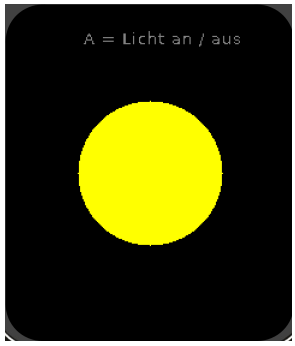
k08_loesung4.py

```
import pbc
import time

an = False
pbc.clear()
pbc.text12("A = Licht an / aus", 65, 20, pbc.GRAY)
pbc.ellipse(120, 140, 60, 60, pbc.GRAY, fill=True)

while True:
    if pbc.was_pressed_a():
        an = not an
        if an:
            pbc.ellipse(120, 140, 60, 60, pbc.YELLOW, fill=True)
        else:
            pbc.ellipse(120, 140, 60, 60, pbc.GRAY, fill=True)
    time.sleep_ms(20)
```

`an = not an` macht aus `False` ein `True` und umgekehrt. Danach entscheidet `if an:`, in welcher Farbe die Lampe gemalt wird. Gezeichnet wird nur, wenn sich etwas ändert, also nur nach einem Druck. Das Bild zeigt die Lampe nach einem Druck auf A.



Aufgabe 5 – Taktgefühl (`k08_loesung5.py`)

k08_loesung5.py

```
import pbc
import time

pbc.clear()
pbc.text12("Druecke A und zaehle", 50, 90, pbc.WHITE)
pbc.text12("im Kopf bis 5.", 70, 110, pbc.WHITE)
pbc.text12("Dann druecke A wieder.", 40, 130, pbc.WHITE)
while not pbc.was_pressed_a():
    time.sleep_ms(10)

start = time.ticks_ms()
pbc.clear(pbc.BLUE)
while not pbc.was_pressed_a():
    time.sleep_ms(10)
zeit = time.ticks_diff(time.ticks_ms(), start)

abweichung = zeit - 5000
if abweichung < 0:
    abweichung = -abweichung
```



```
20 print(f"Gemessen: {zeit} ms, daneben: {abweichung} ms")
21
22 pbc.clear()
23 pbc.text24(f"{zeit} ms", 70, 90, pbc.WHITE)
24 if abweichung < 200:
25     pbc.text18("Wie eine Uhr!", 60, 140, pbc.GREEN)
26 elif abweichung < 1000:
27     pbc.text18("Nicht schlecht", 55, 140, pbc.YELLOW)
28 else:
29     pbc.text18("Uebe noch!", 75, 140, pbc.RED)
```

Terminal

```
Gemessen: 5302 ms, daneben: 302 ms
```

- Die Messung (Zeilen 8 bis 15) funktioniert wie bei der Stoppuhr in 8.5.
- `zeit - 5000` ist negativ, wenn zu früh gedrückt wurde. Die Zeilen 18 und 19 machen daraus eine positive Zahl. So muss die Bewertung nur noch prüfen, wie weit man danebenlag, egal in welche Richtung.
- Die Bewertung prüft mit `if` / `elif` / `else` von der kleinsten Grenze aus.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 9: LEDs und Töne

Aufgabe 1 – Welche Lichter brennen?



Nach einer halben Sekunde leuchten die rote LED (Zeile 4, `pbc.led_red()` ohne Wert bedeutet `True`) und die grüne LED (Zeile 5). Die RGB-LED leuchtet blau (Zeile 6).

Nach dem Programmende ist Rot aus (Zeile 8), Grün aus (Zeile 12), aber Gelb an (Zeile 9). Die RGB-LED leuchtet gelb, denn `255, 255, 0` mischt Rot und Grün (Zeile 10). Beide bleiben an, weil das Programm sie nicht ausschaltet. Im Emulator gehen sie erst beim nächsten ▶ **Testen** oder bei ■ **Stop** aus.

Aufgabe 2 – Finde den Fehler (`k09_loesung2.py`)

Man hört nur **einen** kurzen Ton statt zwei. `pbc.tone` wartet nicht: Zeile 6 ersetzt den Ton aus Zeile 5 sofort. Hinter den ersten kurzen Ton gehört eine Pause, die etwas länger ist als der Ton:

k09_loesung2.py

```
 1 import pbc
 2 import time
 3
 4 # Signal: kurz - kurz - lang
 5 pbc.tone(880, 150)
 6 time.sleep_ms(250)
 7 pbc.tone(880, 150)
 8 time.sleep_ms(250)
 9 pbc.tone(660, 1000)
10 time.sleep_ms(1000)
```

Die Pausen sind 100 ms länger als die kurzen Töne. So entsteht zwischen den Tönen eine hörbare Lücke.

Aufgabe 3 – Die Entchen schwimmen weiter (`k09_loesung3.py`)

k09_loesung3.py

```
 1 import pbc
 2 import time
 3
 4 C = 262
 5 D = 294
 6 E = 330
 7 F = 349
 8 G = 392
 9 A = 440
10
11 VIERTEL = 300
12 HALBE = 600
13
14 # Alle meine Entchen ...
15 pbc.tone(C, VIERTEL - 50)
16 time.sleep_ms(VIERTEL)
17 pbc.tone(D, VIERTEL - 50)
18 time.sleep_ms(VIERTEL)
19 pbc.tone(E, VIERTEL - 50)
20 time.sleep_ms(VIERTEL)
21 pbc.tone(F, VIERTEL - 50)
22 time.sleep_ms(VIERTEL)
23 pbc.tone(G, HALBE - 50)
24 time.sleep_ms(HALBE)
25 pbc.tone(G, HALBE - 50)
26 time.sleep_ms(HALBE)
```

```
28 # ... schwimmen auf dem See: viermal A, dann G
29 for note in range(4):
30     pbc.tone(A, VIERTEL - 50)
31     time.sleep_ms(VIERTEL)
32 pbc.tone(G, 2 * HALBE - 50)
33 time.sleep_ms(2 * HALBE)
```

Neu ist die Konstante `A = 440` (Zeile 9). Die vier gleichen Töne spielt eine `for`-Schleife (Zeile 29 bis 31). Die letzte Note ist so lang wie zwei halbe Noten, also `2 * HALBE`.

Aufgabe 4 – Blaulicht (`k09_loesung4.py`)

k09_loesung4.py

```
 1 import pbc
 2 import time
 3
 4 pbc.clear()
 5 pbc.text24("Einsatz!", 70, 120, pbc.WHITE)
 6
 7 for runde in range(5):
 8     pbc.rgb_led(0, 0, 255)       # blau
 9     pbc.tone(440)
10     time.sleep_ms(500)
11     pbc.rgb_led(255, 0, 0)       # rot
12     pbc.tone(587)
13     time.sleep_ms(500)
14
15 pbc.tone_off()
16 pbc.rgb_led_off()
```



Hier passen Dauertöne: Jeder neue Ton ersetzt den vorigen, und die Pause von 500 ms bestimmt, wie lange er klingt. Nach der Schleife schalten Zeile 15 und 16 Ton und Licht aus. Ohne Zeile 15 würde der letzte Ton immer weiter klingen.

Aufgabe 5 – Ein Metronom (`k09_loesung5.py`)

k09_loesung5.py

```
 1 import pbc
 2 import time
 3
 4 tempo = 100     # Schlaege pro Minute
 5 neu = True      # muss die Zahl neu gezeichnet werden?
 6
 7 pbc.clear()
 8 pbc.text18("Metronom", 80, 40, pbc.WHITE)
 9 pbc.text12("Schlaege pro Minute", 55, 160, pbc.WHITE)
10 pbc.text12("hoch = schneller", 20, 220, pbc.GRAY)
11 pbc.text12("runter = langsamer", 20, 240, pbc.GRAY)
```



```
while True:
    if pbc.was_pressed_up() and tempo < 200:
        tempo = tempo + 10
        neu = True
    if pbc.was_pressed_down() and tempo > 40:
        tempo = tempo - 10
        neu = True
    if neu:
        pbc.fill_rect(60, 105, 120, 35, pbc.BLACK)
        pbc.text24(f"{tempo}", 100, 110, pbc.YELLOW)
        neu = False

    # ein Schlag: Klick und rote LED
    pause = 60000 // tempo
    pbc.tone(1000, 30)
    pbc.led_red(True)
    time.sleep_ms(80)
    pbc.led_red(False)
    time.sleep_ms(pause - 80)
```

Eine Minute hat 60 000 ms. Bei 100 Schlägen pro Minute liegen 60 000 // 100 = **600 ms** zwischen zwei Klicks. Ein Schlag besteht aus 80 ms mit leuchtender LED und `pause - 80` ms Warten; zusammen sind das genau `pause` Millisekunden.

`was_pressed_up()` sorgt dafür, dass ein Druck das Tempo nur einmal um 10 erhöht. Das `and tempo < 200` begrenzt das Tempo nach oben, sonst würde `pause - 80` irgendwann kleiner als null. Die Variable `neu` merkt sich, ob sich das Tempo geändert hat. Nur dann wird die alte Zahl mit einem schwarzen Rechteck übermalt und die neue geschrieben. So flackert nichts, und wenn die Zahl von 100 auf 90 schrumpft, bleibt keine alte Ziffer stehen.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 10: Texte (Strings)

Aufgabe 1 – Was gibt das Programm aus?

Terminal

```
7
ag
HAM
Hammer
True False
```

- `"Hamburg"` hat 7 Zeichen.
- `stadt[1]` ist `"a"` (Index 1 ist das **zweite** Zeichen), `stadt[-1]` ist `"g"`. Mit `+` werden sie zu `"ag"` verbunden.
- `stadt[0:3]` ist `"Ham"`, in Großbuchstaben `"HAM"`.
- `replace` ersetzt `"burg"` durch `"mer"`: `"Hammer"`.

- `"b"` kommt vor, `"B"` nicht, denn Groß- und Kleinschreibung zählen.

Aufgabe 2 – Finde die Fehler (`k10_loesung2.py`)

1. Zeile 2: `IndexError: string index out of range`. `"banane"` hat 6 Zeichen mit den Indizes 0 bis 5. Richtig ist `frucht[5]` oder `frucht[-1]`.
2. Danach Zeile 4: `TypeError: 'str' object doesn't support item assignment`. Ein Zeichen in einem String kann man nicht ersetzen. Stattdessen baut man einen neuen Text aus `"B"` und dem Rest ab Index 1.

k10_loesung2.py

```
frucht = "banane"
letztes = frucht[5]          # oder frucht[-1]
print("Letzter Buchstabe:", letztes)
frucht = "B" + frucht[1:]
print(frucht)
```

Terminal

```
Letzter Buchstabe: e
Banane
```

Aufgabe 3 – Rückwärts und Palindrome (`k10_loesung3.py`)

k10_loesung3.py

```
wort = input("Wort: ")

rueckwaerts = ""
for zeichen in wort:
    rueckwaerts = zeichen + rueckwaerts
print("Rückwärts:", rueckwaerts)

if wort.lower() == rueckwaerts.lower():
    print("Das ist ein Palindrom!")
else:
    print("Kein Palindrom.")
```

Terminal

```
Wort: Rentner
Rückwärts: rentneR
Das ist ein Palindrom!
```

`zeichen + rueckwaerts` setzt das neue Zeichen **vor** den bisherigen Text. Aus „Rentner“ wird so nacheinander „R“, „eR“, „neR“ … bis „rentneR“. Für den Vergleich werden beide Texte mit `lower()` kleingeschrieben, sonst wäre „Rentner“ wegen des großen R kein Palindrom.

Aufgabe 4 – Melodie als Text (`k10_loesung4.py`)

k10_loesung4.py

```
 1 import pbc
 2 import time
 3
 4 melodie = "cdefgg-aaaag---"   # "-" ist eine Pause
 5 DAUER = 300
 6
 7 for note in melodie:
 8     frequenz = 0
 9     if note == "c":
10         frequenz = 262
11     elif note == "d":
12         frequenz = 294
13     elif note == "e":
14         frequenz = 330
15     elif note == "f":
16         frequenz = 349
17     elif note == "g":
18         frequenz = 392
19     elif note == "a":
20         frequenz = 440
21     elif note == "h":
22         frequenz = 494
23     if frequenz > 0:
24         pbc.tone(frequenz, DAUER - 50)
25     time.sleep_ms(DAUER)
```

Für jeden Buchstaben sucht die `if`-`elif`-Kette die Frequenz heraus. Bei `-` (und jedem unbekannten Zeichen) bleibt `frequenz` 0; dann spielt Zeile 24 keinen Ton, aber Zeile 25 wartet trotzdem: Das ist die Pause. Eine lange Note entsteht hier aus mehreren Pausen hinter dem Ton. Ein neues Lied brauchst du jetzt nur noch als Text in Zeile 4 einzutragen.

Aufgabe 5 – Benutzername (`k10_loesung5.py`)

k10_loesung5.py

```
1 vorname = input("Vorname: ")
2 nachname = input("Nachname: ")
3
4 benutzer = vorname[:3].lower() + nachname[:3].lower()
5 print("Dein Benutzername:", benutzer)
6 print("Er hat", len(benutzer), "Zeichen.")
```

Terminal

```
Vorname: Lina
Nachname: Schröder
Dein Benutzername: linsch
Er hat 6 Zeichen.
```

`vorname[:3]` sind die ersten drei Zeichen. Ist ein Name kürzer, etwa „Al“, gibt es keinen Fehler; das Stück ist dann einfach kürzer.

Aufgabe 6 – Den Code knacken (`k10_loesung6.py`)

k10_loesung6.py

```
geheim = "WUHIISXQNW DP EUXQQHQ"

for schluessel in range(1, 26):
    klar = ""
    for zeichen in geheim:
        if "A" <= zeichen <= "Z":
            nummer = (ord(zeichen) - ord("A") - schluessel) % 26
            klar += chr(nummer + ord("A"))
        else:
            klar += zeichen
    print(schluessel, klar)
```

Terminal

```
1 VTGHHRWPMV CO DTWPPGP
2 USFGGQVOLU BN CSVOOFO
3 TREFFPUNKT AM BRUNNEN
4 SQDEEOTMJS ZL AQTMMDM
5 RPCDDNSLIR YK ZPSLLCL
6 QOBCCMRKHQ XJ YORKKBK
7 PNABBLQJGP WI XNQJJAJ
8 OMZAAKPIFO VH WMPIIZI
9 NLYZZJOHEN UG VLOHHYH
10 MKXYYINGDM TF UKNGGXG
11 LJWXXHMFCL SE TJMFFWF
12 KIVWWGLEBK RD SILEEVE
13 JHUVVFKDAJ QC RHKDDUD
14 IGTUUEJCZI PB QGJCCTC
15 HFSTTDIBYH OA PFIBBSB
16 GERSSCHAXG NZ OEHAARA
17 FDQRRBGZWF MY NDGZZQZ
18 ECPQQAFYVE LX MCFYYPY
19 DBOPPZEXUD KW LBEXXOX
20 CANOOYDWTC JV KADWWNW
21 BZMNNXCVSB IU JZCVVMV
22 AYLMMWBURA HT IYBUULU
23 ZXKLLVATQZ GS HXATTKT
24 YWJKKUZSPY FR GWZSSJS
25 XVIJJTYROX EQ FVYRRIR
```

Die innere Schleife ist das Programm aus „Ausprobieren“, nur mit `-` statt `+`. Die äußere Schleife probiert alle 25 Schlüssel. In Zeile 3 steht „TREFFPUNKT AM BRUNNEN“, der Schlüssel war also 3. Die Caesar-Verschlüsselung ist sehr unsicher: Es gibt nur 25 Schlüssel, und ein Computer probiert sie in einem Augenblick alle durch.

Lösungen zu Kapitel 11: Funktionen

Aufgabe 1 – Was gibt das Programm aus?

```
Terminal
Start
B eins
A
A
B zwei
A
Ende
```

Die beiden `def` geben nichts aus. Danach läuft das Hauptprogramm: `Start`, dann ruft `b("eins")` erst `print("B", text)` auf und anschließend `a()`. Darum folgt auf jedes `B …` ein `A`. Der einzelne Aufruf `a()` dazwischen erzeugt das zweite `A`.

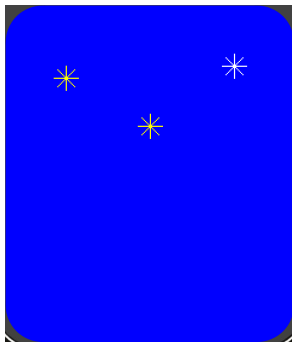
Aufgabe 2 – Finde die Fehler (`k11_loesung2.py`)

Python meldet die Fehler einzeln, nacheinander:

1. `SyntaxError` in Zeile 4: Hinter `def stern(x, y, farbe)` fehlt der Doppelpunkt.
2. Nach der Korrektur: `TypeError` in Zeile 13: `stern(120, 100)` hat nur zwei Argumente, die Funktion erwartet drei. Die Farbe fehlt.
3. Danach: `NameError` in Zeile 14: `Stern` mit großem S gibt es nicht. Die Funktion heißt `stern`.

k11_loesung2.py

```
 1 import pbc
 2
 3
 4 def stern(x, y, farbe):
 5     pbc.line(x - 10, y, x + 10, y, farbe)
 6     pbc.line(x, y - 10, x, y + 10, farbe)
 7     pbc.line(x - 7, y - 7, x + 7, y + 7, farbe)
 8     pbc.line(x - 7, y + 7, x + 7, y - 7, farbe)
 9
10
11 pbc.clear(pbc.BLUE)
12 stern(50, 60, pbc.YELLOW)
13 stern(120, 100, pbc.YELLOW)
14 stern(190, 50, pbc.WHITE)
```



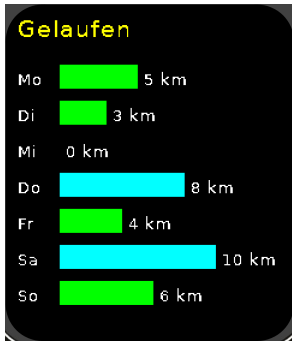
Aufgabe 3 – Das Lauftagebuch (`k11_loesung3.py`)

k11_loesung3.py

```
import pbc


def balken(y, tag, km, farbe):
    pbc.text12(tag, 10, y + 4, pbc.WHITE)
    pbc.fill_rect(45, y, km * 13, 20, farbe)
    pbc.text12(f"{km} km", 50 + km * 13, y + 4, pbc.WHITE)


pbc.clear()
pbc.text18("Gelaufen", 10, 10, pbc.YELLOW)
balken(50, "Mo", 5, pbc.GREEN)
balken(80, "Di", 3, pbc.GREEN)
balken(110, "Mi", 0, pbc.GREEN)
balken(140, "Do", 8, pbc.CYAN)
balken(170, "Fr", 4, pbc.GREEN)
balken(200, "Sa", 10, pbc.CYAN)
balken(230, "So", 6, pbc.GREEN)
```



Ein Kilometer ergibt 13 Pixel Balken. So passt auch ein Lauf von 10 km mit Beschriftung auf den Bildschirm. Die Beschriftung beginnt 5 Pixel hinter dem Ende des Balkens (`50 + km * 13`). Bei 0 km ist der Balken 0 Pixel breit und nicht zu sehen. Längere Läufe sind zur Hervorhebung cyan.

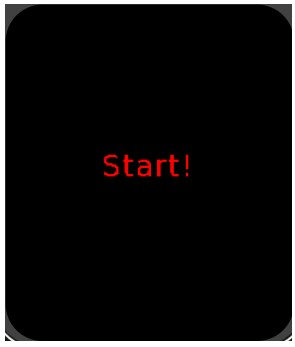
Aufgabe 4 – Ein besserer Countdown (`k11_loesung4.py`)

k11_loesung4.py

```
import pbc
import time


def countdown(von, farbe=pbc.WHITE, ziel="Los!"):
    for zahl in range(von, 0, -1):
        pbc.clear()
        pbc.text24(str(zahl), 112, 120, farbe)
        pbc.tone(440, 100)
        time.sleep(1)
    pbc.clear()
    pbc.text24(ziel, 80, 120, farbe)
    pbc.tone(880, 400)


countdown(3)
time.sleep(1)
countdown(5, pbc.RED, "Start!")
```



`countdown(3)` benutzt beide Standardwerte: weiße Zahlen, am Ende „Los!“. Der zweite Aufruf gibt alle drei Argumente an. Der Ton in Zeile 10 dauert 100 ms und klingt, während `time.sleep(1)` wartet, denn `pbc.tone` wartet nicht. Der Text am Ende steht bei x = 80, damit auch das längere

„Start!“ ungefähr in der Mitte steht.

Aufgabe 5 – Flaggen (`k11_loesung5.py`)

k11_loesung5.py

```
import pbc


def flagge(x, y, f1, f2, f3, quer=True):
    if quer:
        # drei Streifen untereinander
        pbc.fill_rect(x, y, 90, 18, f1)
        pbc.fill_rect(x, y + 18, 90, 18, f2)
        pbc.fill_rect(x, y + 36, 90, 18, f3)
    else:
        # drei Streifen nebeneinander
        pbc.fill_rect(x, y, 30, 54, f1)
        pbc.fill_rect(x + 30, y, 30, 54, f2)
        pbc.fill_rect(x + 60, y, 30, 54, f3)


def beschriftet(name, x, y, f1, f2, f3, quer=True):
    flagge(x, y, f1, f2, f3, quer)
    pbc.text(name, x, y + 60, pbc.WHITE)


pbc.clear(pbc.GRAY)
beschriftet("Deutschland", 20, 20, pbc.BLACK, pbc.RED, pbc.YELLOW)
beschriftet("Oesterreich", 130, 20, pbc.RED, pbc.WHITE, pbc.RED)
beschriftet("Frankreich", 20, 105, pbc.BLUE, pbc.WHITE, pbc.RED, False)
beschriftet("Italien", 130, 105, pbc.GREEN, pbc.WHITE, pbc.RED, False)
beschriftet("Niederlande", 20, 190, pbc.RED, pbc.WHITE, pbc.BLUE)
beschriftet("Irland", 130, 190, pbc.GREEN, pbc.WHITE, pbc.ORANGE, False)
```



`flagge` entscheidet mit `if quer:`, ob die Streifen übereinander (je 18 Pixel hoch) oder nebeneinander (je 30 Pixel breit) liegen. Weil `quer` den Standardwert `True` hat, müssen nur die Flaggen mit senkrechten Streifen `False` angeben. `beschriftet` reicht ihre Parameter einfach an `flagge` weiter und schreibt den Namen darunter. Der Hintergrund ist grau, damit der schwarze Streifen der deutschen Flagge zu sehen ist. Gold gibt es nicht als Farbkonstante; `pbc.YELLOW` kommt ihm am nächsten.

Lösungen zu Kapitel 12: Funktionen mit Rückgabewert

Aufgabe 1 – Wie schnell? (`k12_loesung1.py`)

k12_loesung1.py

```
def kmh(meter, sekunden):
    return meter / sekunden * 3.6


# Tests
print(kmh(100, 10), "erwartet: 36.0")
assert kmh(100, 10) == 36
assert kmh(1000, 3600) == 1

print("100 m in 12,5 s:", kmh(100, 12.5), "km/h")
marathon = kmh(42195, 2 * 3600 + 1 * 60 + 9)
print("Marathon-Weltrekord:", round(marathon, 1), "km/h")
```

Terminal

```
36.0 erwartet: 36.0
100 m in 12,5 s: 28.8 km/h
Marathon-Weltrekord: 20.9 km/h
```

Meter pro Sekunde mal 3,6 ergibt km/h. Die Marathonzeit wird in Sekunden umgerechnet, bevor sie in die Funktion geht. `round(…, 1)` rundet auf eine Nachkommastelle. Der Test `kmh(100, 10) == 36` ist bestanden, obwohl die Funktion `36.0` liefert: Python hält 36 und 36.0 für gleich.

Aufgabe 2 – Was gibt das Programm aus?

Terminal

```
Wert: 26
Wert: 5
None
9
```

- `a` ist `dreifach(4)` = 12, `b` ist `dreifach(12) - 10` = 26. `zeige(b)` gibt `Wert: 26` aus.
- `c = zeige(5)` gibt `Wert: 5` aus. `zeige` hat kein `return`, also ist `c` gleich `None`.
- `dreifach(dreifach(1))`: Erst wird der innere Aufruf ausgerechnet (3), dann der äußere mit diesem Wert (9).

Aufgabe 3 – Finde die Fehler (`k12_loesung3.py`)

Erster Testlauf:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 15, in <module>
  File "<stdin>", line 5, in verliere_leben
NameError: local variable referenced before assignment
```

1. **`global` fehlt.** `verliere_leben` verändert `leben`, deshalb ist `leben` dort lokal und hat in Zeile 5 noch keinen Wert. Abhilfe: `global leben` als erste Zeile der Funktion.
2. **`print` statt `return`.** Danach läuft das Programm ohne Fehlermeldung, gibt aber nur `True` aus und nie „Game over“: `ist_vorbei` *zeigt* das Ergebnis an, gibt aber `None` zurück, und `if None:` gilt als falsch. Abhilfe: `return leben == 0`.

k12_loesung3.py

```
leben = 3


def verliere_leben():
    global leben
    leben = leben - 1


def ist_vorbei():
    return leben == 0


verliere_leben()
verliere_leben()
verliere_leben()
if ist_vorbei():
    print("Game over")
```

Terminal

```
Game over
```

Aufgabe 4 – Vokale zählen (`k12_loesung4.py`)

k12_loesung4.py

```
def ist_vokal(zeichen):
    return zeichen in "aeiouäöüAEIOUÄÖÜ"


def zaehle_vokale(text):
    anzahl = 0
    for zeichen in text:
        if ist_vokal(zeichen):
            anzahl += 1
```

```
    return anzahl


assert ist_vokal("a") and ist_vokal("Ü")
assert not ist_vokal("b") and not ist_vokal(" ")
assert zaehle_vokale("Hallo") == 2
assert zaehle_vokale("") == 0
print(zaehle_vokale("Programmieren macht Spaß"))
```

Terminal

```
7
```

`zeichen in "aeiou…"` ist schon `True` oder `False` und kann direkt zurückgegeben werden. `zaehle_vokale` ruft `ist_vokal` für jedes Zeichen auf. Der Test mit dem leeren Text prüft den Grenzfall: Die Schleife läuft dann gar nicht, und die Funktion gibt `0` zurück. In „Programmieren macht Spaß“ stecken o, a, i, e, e, a, a: 7 Vokale.

Aufgabe 5 – Erst die Tests, dann die Funktion (`k12_loesung5.py`)

Mit der unfertigen Funktion scheitert schon der erste Test:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
AssertionError:
```

k12_loesung5.py

```
def quersumme(zahl):
    summe = 0
    while zahl > 0:
        summe += zahl % 10      # letzte Ziffer
        zahl = zahl // 10       # letzte Ziffer abschneiden
    return summe


assert quersumme(7) == 7
assert quersumme(123) == 6
assert quersumme(9999) == 36
assert quersumme(1000) == 1
assert quersumme(0) == 0
print("Alle Tests bestanden")
```

Terminal

```
Alle Tests bestanden
```

Die Schleife holt mit `% 10` die letzte Ziffer, addiert sie und schneidet sie mit `// 10` ab, bis nichts mehr übrig ist. Für die Zahl 0 läuft die Schleife gar nicht; die Quersumme ist dann 0, wie im letzten Test verlangt.

Aufgabe 6 – Pasch! (`k12_loesung6.py`)

k12_loesung6.py

```
1  import random
2
3
4  def wuerfle_zwei():
5      return random.randint(1, 6), random.randint(1, 6)
6
7
8  def ist_pasch(a, b):
9      return a == b
10
11
12 versuche = 0
13 pasch = False
14 while not pasch:
15     w1, w2 = wuerfle_zwei()
16     versuche += 1
17     print(f"Wurf {versuche}: {w1} und {w2}")
18     pasch = ist_pasch(w1, w2)
19 print("Pasch nach", versuche, "Würfen!")
```

Terminal (ein möglicher Ablauf)

```
Wurf 1: 4 und 4
Pasch nach 1 Würfen!
```

`wuerfle_zwei` gibt zwei Zufallszahlen zurück, die in Zeile 15 entpackt werden. Die Variable `pasch` startet mit `False`, damit die Schleife mindestens einmal läuft. Bei jedem Start ist die Ausgabe anders.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 13: Listen

Aufgabe 1 – Was gibt das Programm aus?

Terminal

```
gelb
blau
4
rot ['gelb', 'blau', 'gruen']
['gelb', 'lila', 'blau', 'gruen']
```

Nach `append` hat die Liste vier Elemente: `["rot", "gelb", "blau", "gruen"]`. `farben[1]` ist das **zweite** Element („gelb“), `farben[-2]` das vorletzte („blau“). `pop(0)` nimmt „rot“ heraus, danach rücken alle nach vorn. `insert(1, "lila")` schiebt „lila“ vor das Element mit Index 1.

Aufgabe 2 – Finde den Fehler (`k13_loesung2.py`)

Das fehlerhafte Programm gibt Folgendes aus:

Terminal

```
Arbeit 1: Note 3
Arbeit 2: Note 1
Arbeit 3: Note 4
Arbeit 4: Note 2

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
IndexError: list index out of range
```

`range(1, len(noten) + 1)` liefert 1 bis 5. Die Indizes der Liste gehen aber von 0 bis 4. Deshalb fehlt die erste Note (Index 0), und bei `noten[5]` bricht das Programm mit `IndexError` ab. Richtig ist `range(len(noten))`. Die Nummer für den Menschen ist dann `i + 1`:

k13_loesung2.py

```
1 noten = [2, 3, 1, 4, 2]
2
3 for i in range(len(noten)):
4     print(f"Arbeit {i + 1}: Note {noten[i]}")
5
6 print("Schnitt:", sum(noten) / len(noten))
```

Terminal

```
Arbeit 1: Note 2
Arbeit 2: Note 3
Arbeit 3: Note 1
Arbeit 4: Note 4
Arbeit 5: Note 2
Schnitt: 2.4
```

Aufgabe 3 – Würfelstatistik (`k13_loesung3.py`)

k13_loesung3.py

```
1  import random
2
3  wuerfe = []
4  for i in range(30):
5      wuerfe.append(random.randint(1, 6))
6  print(wuerfe)
7
8  for augen in range(1, 7):
9      anzahl = wuerfe.count(augen)
10     print(f"{augen}: {anzahl} mal")
11
12 print("Durchschnitt:", round(sum(wuerfe) / len(wuerfe), 2))
```

Terminal

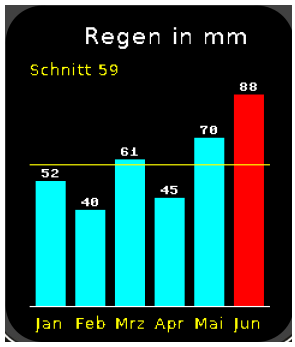
```
[3, 1, 3, 3, 2, 3, 5, 6, 1, 2, 6, 2, 4, 6, 5, 3, 5, 2, 4, 6, 2, 1, 1, 6, 4, 3, 1, 2,
  1, 3]
1: 6 mal
2: 6 mal
3: 7 mal
4: 3 mal
5: 3 mal
6: 5 mal
Durchschnitt: 3.2
```

Die Liste beginnt leer und wächst mit jedem `append` um einen Wurf. Die zweite Schleife läuft über die Augenzahlen 1 bis 6 und zählt jede mit `count`. Bei jedem Start fallen andere Zahlen. Je öfter man würfelt, desto näher liegt der Durchschnitt bei 3,5.

Aufgabe 4 – Das Diagramm verbessern (`k13_loesung4.py`)

k13_loesung4.py

```
 1 import pbc
 2
 3 # Regenmenge in mm, Januar bis Juni
 4 monate = ["Jan", "Feb", "Mrz", "Apr", "Mai", "Jun"]
 5 regen = [52, 40, 61, 45, 70, 88]
 6 groesster = max(regen)
 7 schnitt = sum(regen) / len(regen)
 8
 9 pbc.clear()
10 pbc.text18("Regen in mm", 65, 15, pbc.WHITE)
11 pbc.hline(20, 250, 200, pbc.WHITE)
12
13 for i in range(len(regen)):
14     x = 25 + i * 33
15     hoehe = regen[i] * 2
16     farbe = pbc.CYAN
17     if regen[i] == groesster:
18         farbe = pbc.RED
19     pbc.fill_rect(x, 250 - hoehe, 25, hoehe, farbe)
20     pbc.text(str(regen[i]), x + 4, 240 - hoehe, pbc.WHITE)
21     pbc.text12(monate[i], x, 256, pbc.YELLOW)
22
23 # Durchschnitt als gelbe Linie
24 y = 250 - int(schnitt * 2)
25 pbc.hline(20, y, 200, pbc.YELLOW)
26 pbc.text12(f"Schnitt {round(schnitt)}", 20, 45, pbc.YELLOW)
```



`max(regen)` wird einmal vor der Schleife berechnet (Zeile 6). In der Schleife vergleicht Zeile 17 jeden Wert damit. Die Linie für den Durchschnitt liegt bei `250 - schnitt * 2`, genau wie die Oberkante eines Balkens. `int` macht daraus eine ganze Zahl, denn Pixel-Koordinaten müssen ganze Zahlen sein.

Aufgabe 5 – Eigene Auswertungen (`k13_loesung5.py`)

k13_loesung5.py

```python
def spannweite(werte):
    return max(werte) - min(werte)

def ueber_durchschnitt(werte):
    schnitt = sum(werte) / len(werte)
    ergebnis = []
    for wert in werte:
        if wert > schnitt:
            ergebnis.append(wert)
    return ergebnis

# Testen mit Werten, deren Ergebnis man im Kopf kennt
print(spannweite([3, 9, 5]))
print(ueber_durchschnitt([1, 2, 3, 4, 5]))

temperaturen = [14, 17, 21, 19, 12, 16]
print("Spannweite:", spannweite(temperaturen))
print("Ueber dem Schnitt:", ueber_durchschnitt(temperaturen))
```

Terminal

```
6
[4, 5]
Spannweite: 9
Ueber dem Schnitt: [17, 21, 19]
```

Die ersten beiden Tests kann man im Kopf nachprüfen: 9 − 3 = 6, und der Durchschnitt von 1 bis 5 ist 3, darüber liegen 4 und 5. Erst danach wird die Funktion mit echten Daten benutzt.

Aufgabe 6 – Notenspiegel (`k13_loesung6.py`)

k13_loesung6.py

```python
import pbc

noten = []
while True:
    eingabe = input("Note (leer = fertig): ")
    if eingabe == "":
        break
    noten.append(int(eingabe))

pbc.clear()
pbc.text18("Notenspiegel", 60, 15, pbc.WHITE)
for note in range(1, 7):
    anzahl = noten.count(note)
    print(f"Note {note}: {anzahl}")
    y = 50 + (note - 1) * 35
    pbc.text18(str(note), 20, y, pbc.YELLOW)
```

```
    pbc.fill_rect(45, y, anzahl * 25, 22, pbc.GREEN)
    pbc.text12(str(anzahl), 50 + anzahl * 25, y + 4, pbc.WHITE)

print("Schnitt:", round(sum(noten) / len(noten), 2))
```

Die Eingaben werden zuerst vollständig in der Liste `noten` gesammelt. Erst danach wird ausgewertet: `noten.count(note)` liefert für jede Note die Anzahl, und jeder Balken ist 25 Pixel pro Note lang. Eine Note, die keiner hat, bekommt einen Balken der Breite 0 – er ist unsichtbar.



Getestet mit den Eingaben 2, 3, 1, 2, 4, 3, 2, 5, 3, 2:

Terminal

```
Note (leer = fertig): 2
Note (leer = fertig): 3
Note (leer = fertig): 1
Note (leer = fertig): 2
Note (leer = fertig): 4
Note (leer = fertig): 3
Note (leer = fertig): 2
Note (leer = fertig): 5
Note (leer = fertig): 3
Note (leer = fertig): 2
Note (leer = fertig):
Note 1: 1
Note 2: 4
Note 3: 3
Note 4: 1
Note 5: 1
Note 6: 0
Schnitt: 2.7
```

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 14: Listen im Spiel

Aufgabe 1 – Was gibt das Programm aus?

Terminal

```
[4, 5, 6]
7
3
3 3
15
[1, 4, 7]
```

`feld[1]` ist die ganze zweite Zeile. `feld[2][0]` ist in Zeile 2 das Element 0, `feld[0][-1]` das letzte Element der ersten Zeile. `len(feld)` zählt die Zeilen, `len(feld[0])` die Elemente einer Zeile. Die Schleife addiert aus jeder Zeile das Element 1: 2 + 5 + 8 = 15. Die List Comprehension sammelt aus jeder Zeile das Element 0, also die erste Spalte.

Aufgabe 2 – Finde den Fehler (`k14_loesung2.py`)

Das fehlerhafte Programm gibt `[5, 7, 0, 3]` aus. Beim Entfernen der ersten Null rücken alle Elemente dahinter eins nach vorn. Die zweite Null landet dadurch auf dem Index, den die Schleife gerade erledigt hat, und wird übersprungen. Zwei Lösungen:

k14_loesung2.py

```
punkte = [5, 0, 0, 7, 0, 3]
for p in punkte[:]:
    if p == 0:
        punkte.remove(p)
print(punkte)

# Oder: eine neue Liste nur mit den gewuenschten Elementen
punkte = [5, 0, 0, 7, 0, 3]
punkte = [p for p in punkte if p != 0]
print(punkte)
```

Terminal

```
[5, 7, 3]
[5, 7, 3]
```

Oben läuft die Schleife über die Kopie `punkte[:]`, entfernt wird im Original. Unten entsteht eine neue Liste, in die nur die Werte ungleich 0 kommen. Auch der Weg rückwärts mit Index und `pop(i)` ist richtig.

Aufgabe 3 – Schneefall (`k14_loesung3.py`)

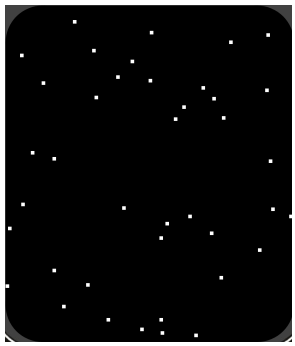
k14_loesung3.py

```
import pbc
import time
import random

ANZAHL = 40
xs = [random.randint(0, 237) for i in range(ANZAHL)]
ys = [random.randint(0, 277) for i in range(ANZAHL)]
tempo = [random.randint(1, 4) for i in range(ANZAHL)]

pbc.clear()
while True:
    for i in range(ANZAHL):
        pbc.fill_rect(xs[i], ys[i], 3, 3, pbc.BLACK)
        ys[i] += tempo[i]
        if ys[i] > 277:                  # unten raus: oben neu
            ys[i] = 0
            xs[i] = random.randint(0, 237)
        pbc.fill_rect(xs[i], ys[i], 3, 3, pbc.WHITE)
    time.sleep_ms(30)
```

Drei parallele Listen, jede mit einer List Comprehension erzeugt. Flocke Nummer `i` hat die Werte `xs[i]`, `ys[i]` und `tempo[i]`. Weil jede Flocke ihr eigenes Tempo hat, fällt der Schnee unterschiedlich schnell; das wirkt räumlich. Ist eine Flocke unten angekommen, wird sie nicht entfernt, sondern wiederverwendet: Sie springt nach oben an eine neue x-Position.



Aufgabe 4 – Durchs Labyrinth (`k14_loesung4.py`)

Die Karte enthält jetzt drei Münzen (Zahl 3), `FARBEN` hat dafür ein viertes Element `pbc.ORANGE`. Neu ab Zeile 24:

k14_loesung4.py

```
24 alle = 0
25 for zeile in range(len(karte)):
26     for spalte in range(len(karte[zeile])):
27         zeichne_feld(zeile, spalte, FARBEN[karte[zeile][spalte]])
28     alle += karte[zeile].count(3)
29
30 z = 1
31 s = 1
32 muenzen = 0
33 zeichne_feld(z, s, pbc.RED)
34 while karte[z][s] != 2:
35     neu_z = z
36     neu_s = s
37     if pbc.was_pressed_up():
38         neu_z -= 1
39     if pbc.was_pressed_down():
40         neu_z += 1
41     if pbc.was_pressed_left():
42         neu_s -= 1
43     if pbc.was_pressed_right():
44         neu_s += 1
45     if karte[neu_z][neu_s] != 1:
46         zeichne_feld(z, s, pbc.BLACK)
47         z = neu_z
48         s = neu_s
49         if karte[z][s] == 3:          # Muenze einsammeln
50             muenzen += 1
51             karte[z][s] = 0
52         zeichne_feld(z, s, pbc.RED)
53     time.sleep_ms(20)
54
55 pbc.text24("Geschafft!", 60, 115, pbc.WHITE, bg=pbc.BLACK)
56 pbc.text18(f"Muenzen: {muenzen} von {alle}", 45, 145, pbc.WHITE, bg=pbc.BLACK)
```

- Zeile 28 zählt beim Zeichnen gleich mit, wie viele Münzen es gibt.
- Zeile 35–44: Erst wird die **neue** Position ausgerechnet. `was_pressed_…` liefert genau einmal pro Druck `True`, die Figur springt also ein Feld weit.
- Zeile 45: Nur wenn dort keine Mauer ist, zieht die Figur um. Weil die Karte ringsum aus Mauern besteht, kann sie nie aus der Liste laufen.
- Zeile 49–51: Eine Münze wird gezählt und in der Karte durch 0 ersetzt, damit sie nicht ein zweites Mal zählt.
- Zeile 34: Die Schleife endet, sobald die Figur auf dem Ziel (2) steht.



Aufgabe 5 – Listen in einer Zeile (`k14_loesung5.py`)

k14_loesung5.py

```
 1 namen = ["Ida", "Konstantin", "Ben", "Marie", "Lu"]
 2
 3 quadrate = [i * i for i in range(1, 11)]
 4 print(quadrate)
 5
 6 laengen = [len(name) for name in namen]
 7 print(laengen)
 8
 9 lange_namen = [name for name in namen if len(name) > 3]
10 print(lange_namen)
11
12 gross = [name.upper() for name in namen]
13 print(gross)
```

Terminal

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[3, 10, 3, 5, 2]
['Konstantin', 'Marie']
['IDA', 'KONSTANTIN', 'BEN', 'MARIE', 'LU']
```

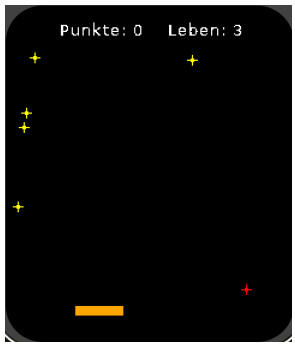
Aufgabe 6 – Meteore im Sternenregen (`k14_loesung6.py`)

Jeder Stern bekommt als viertes Element seine Farbe. Der Anfang (Zeile 1–29) entspricht `k14_sternenregen.py`; jeder Stern ist jetzt `[x, y, tempo, farbe]`.

k14_loesung6.py

```
31     if random.randint(1, 12) == 1:
32         farbe = pbc.YELLOW
33         if random.randint(1, 4) == 1:     # Meteor
34             farbe = pbc.RED
35         tempo = random.randint(2, 5) + punkte // 10
36         sterne.append([random.randint(10, 230), 35, tempo, farbe])
37
38     for stern in sterne[:]:
39         zeichne_stern(stern[0], stern[1], pbc.BLACK)
```

```
40          stern[1] += stern[2]
41          if stern[1] >= SCHALE_Y - 4:
42              sterne.remove(stern)
43              gefangen = schale_x - 4 <= stern[0] and stern[0] <= schale_x + BREITE
                  + 4
44              if stern[3] == pbc.YELLOW and gefangen:
45                  punkte += 1
46              elif stern[3] == pbc.YELLOW or gefangen:
47                  leben -= 1            # Stern verpasst oder Meteor gefangen
48              zeige_stand()
49          else:
50              zeichne_stern(stern[0], stern[1], stern[3])
51      time.sleep_ms(30)
52
53  pbc.text24("Game over", 62, 125, pbc.RED)
```



- Zeile 33–34: `randint(1, 4) == 1` trifft im Schnitt in jedem vierten Fall zu.
- Zeile 35: `punkte // 10` ist bei 0 bis 9 Punkten 0, ab 10 Punkten 1 und so weiter. Neue Sterne werden also alle 10 Punkte schneller.
- Zeile 43: `gefangen` ist `True` oder `False`. Einen Vergleich kann man wie eine Zahl in einer Variablen speichern.
- Zeile 44–47: Ein gelber, gefangener Stern bringt einen Punkt. Sonst kostet es ein Leben, wenn der Stern gelb war (verpasst) oder wenn er gefangen wurde (also ein Meteor war). Ein verpasster Meteor ändert nichts.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 15: Dateien: Daten dauerhaft speichern

Aufgabe 1 – Was steht in der Datei?

Terminal
```
blau
gelb
```

„rot“ und „gruen“ werden zwar geschrieben, aber der zweite `with`-Block öffnet die Datei wieder im Modus `"w"` und löscht dabei den alten Inhalt. Danach steht nur noch „blau“ in der Datei. Der Modus `"a"` hängt „gelb“ hinten an.

Aufgabe 2 – Finde die Fehler (`k15_loesung2.py`)

Python meldet die Fehler nacheinander. Zuerst kommt `TypeError: object with buffer protocol required` in Zeile 4: `write` braucht einen String. Richtig ist `datei.write(str(muenzen))`. Nach dieser Korrektur kommt ein zweiter `TypeError`, diesmal in Zeile 9:

Terminal

```
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
TypeError: unsupported types for __add__: 'str', 'int'
```

`datei.read()` liefert den Text `"7"`. Text und Zahl kann man nicht addieren. Richtig ist `int(datei.read())`.

k15_loesung2.py

```
1 muenzen = 7
2
3 with open("muenzen.txt", "w") as datei:
4     datei.write(str(muenzen))       # Fehler 1: Zahl in Text umwandeln
5
6 with open("muenzen.txt", "r") as datei:
7     gespeichert = int(datei.read())  # Fehler 2: Text in Zahl umwandeln
8
9 print("Mit Bonus:", gespeichert + 3)
```

Terminal

```
Mit Bonus: 10
```

Aufgabe 3 – Lieblingsfilme (`k15_loesung3.py`)

k15_loesung3.py

```
1  # Teil 1: drei Filme abfragen und speichern
2  with open("filme.txt", "w") as datei:
3      for i in range(3):
4          film = input(f"Lieblingsfilm Nr. {i + 1}: ")
5          datei.write(film + "\n")
6
7  # Teil 2: Datei lesen und nummeriert ausgeben
8  print("Deine Lieblingsfilme:")
9  nummer = 1
10 with open("filme.txt", "r") as datei:
11     for zeile in datei:
12         print(f"{nummer}. {zeile.strip()}")
13         nummer += 1
```

Terminal

```
Lieblingsfilm Nr. 1: Paddington
Lieblingsfilm Nr. 2: Toy Story
Lieblingsfilm Nr. 3: Ratatouille
Deine Lieblingsfilme:
1. Paddington
2. Toy Story
3. Ratatouille
```

Die Schleife in Zeile 3–5 schreibt jeden Film mit `\n` in eine eigene Zeile. Beim Lesen entfernt `strip()` das `\n` wieder. Die Variable `nummer` zählt mit, wie in Abschnitt 15.2.

Aufgabe 4 – Willkommen zurück (`k15_loesung4.py`)

k15_loesung4.py

```python
import os
import pbc

DATEI = "name.txt"

if DATEI in os.listdir():
    with open(DATEI, "r") as datei:
        name = datei.read()
    gruss = "Willkommen zurueck,"
else:
    name = input("Wie heißt du? ")
    with open(DATEI, "w") as datei:
        datei.write(name)
    gruss = "Hallo,"

pbc.clear()
pbc.text18(gruss, 30, 100, pbc.WHITE)
pbc.text24(name + "!", 30, 130, pbc.YELLOW)
```



Beim ersten Start gibt es `name.txt` noch nicht: Das Programm fragt im Terminal nach dem Namen, speichert ihn und zeigt „Hallo, Mila!“ (Bild). Bei jedem weiteren Start ist die Datei da, `input` wird übersprungen, und auf dem Display steht „Willkommen zurueck,“ mit dem gespeicherten Namen. Im Terminal geht das ß in „heißt“, auf dem Display schreiben wir „zurueck“.

Aufgabe 5 – Würfelstatistik über viele Starts (`k15_loesung5.py`)

k15_loesung5.py

```python
import os
import random

DATEI = "wuerfel.txt"

# Bisherige Zählung laden: eine Zeile pro Augenzahl
if DATEI in os.listdir():
    anzahl = []
```

```
 9      with open(DATEI, "r") as datei:
10          for zeile in datei:
11              anzahl.append(int(zeile))
12  else:
13      anzahl = [0, 0, 0, 0, 0, 0]
14
15  # 60-mal würfeln; Index 0 steht für die Augenzahl 1
16  for i in range(60):
17      augen = random.randint(1, 6)
18      anzahl[augen - 1] += 1
19
20  with open(DATEI, "w") as datei:
21      for wert in anzahl:
22          datei.write(str(wert) + "\n")
23
24  print("Würfe insgesamt:", sum(anzahl))
25  for augen in range(1, 7):
26      print(augen, ":", anzahl[augen - 1])
```

Ausgabe nach dem **zweiten** Start (die Zahlen sind bei dir anders, weil zufällig gewürfelt wird):

Terminal

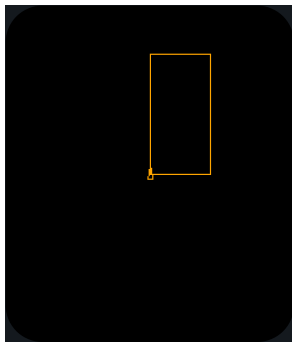
```
Würfe insgesamt: 60
1 : 8
2 : 10
3 : 12
4 : 9
5 : 11
6 : 10
```

- Zeile 7–11: Gibt es die Datei, wird aus den sechs Zeilen eine Liste mit sechs Zahlen. Sonst beginnt die Zählung mit sechs Nullen (Zeile 13).
- Zeile 16–18: `anzahl[0]` zählt die Einsen, `anzahl[5]` die Sechsen. Darum steht dort `augen - 1`.
- Zeile 20–22: Die Liste wird komplett neu geschrieben (Modus `"w"`), eine Zahl pro Zeile.

Je öfter du das Programm startest, desto näher liegen die sechs Anzahlen beieinander: Jede Augenzahl kommt ungefähr in einem Sechstel der Würfe.

Lösungen zu Kapitel 16: Turtle-Grafik

Aufgabe 1 – Was zeichnet die Turtle?



Die Turtle zeichnet ein orangefarbenes Rechteck, 100 Pixel hoch und 50 Pixel breit. Sie startet in der Mitte und schaut nach oben. Die erste Linie geht also nach oben, dann dreht sie sich nach rechts. Das Rechteck liegt deshalb **rechts oberhalb** der Mitte. Jede Runde der Schleife zeichnet zwei Seiten, zwei Runden ergeben vier Seiten. Am Ende hat sich die Turtle viermal um 90 Grad gedreht: Sie steht wieder in der Mitte und schaut nach oben.

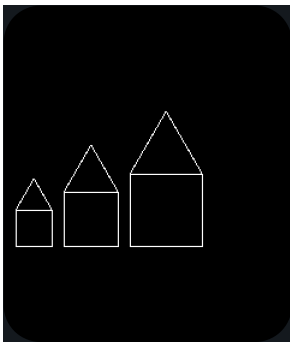
Aufgabe 2 – Häuserreihe (`k16_loesung2.py`)

k16_loesung2.py

```python
import turtle

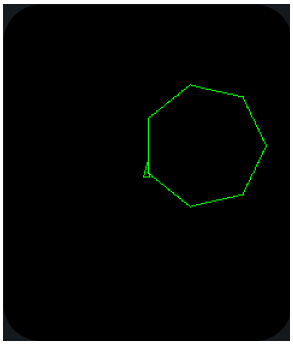

def haus(t, groesse):
    # Wände: ein Quadrat
    for i in range(4):
        t.forward(groesse)
        t.right(90)
    # Dach: ein gleichseitiges Dreieck auf der oberen Kante
    t.forward(groesse)
    t.right(30)
    for i in range(3):
        t.forward(groesse)
        t.right(120)
    t.left(30)                  # wieder nach oben schauen
    t.back(groesse)             # zurück zur Ecke unten links


t = turtle.Turtle()
t.hide()
t.speed(0)
t.setPosition(-110, -60)
for groesse in [30, 45, 60]:
    haus(t, groesse)
    # zum nächsten Haus springen
    t.penUp()
    t.setDirection(90)
    t.forward(groesse + 10)
    t.setDirection(0)
    t.penDown()
```



- Das Quadrat (Zeile 6 bis 8) endet unten links, Blick nach oben.
- Für das Dach läuft die Turtle die linke Wand hinauf (Zeile 10). Nach `right(30)` schaut sie schräg nach rechts oben. Drei Seiten mit 120 Grad ergeben das Dreieck; die dritte Seite liegt genau auf der oberen Kante.
- Danach schaut die Turtle um 30 Grad nach rechts gedreht. `left(30)` richtet sie wieder nach oben aus, `back(groesse)` bringt sie zurück zur Ecke.
- Im Hauptprogramm springt die Turtle mit gehobenem Stift nach rechts zum nächsten Haus: um die Breite des Hauses plus 10 Pixel Abstand.

Aufgabe 3 – Finde die Fehler (`k16_loesung3.py`)



1. `"gruen"` ist kein Farbname, den die Turtle kennt. Unbekannte Namen ergeben ohne Fehlermeldung Weiß. Richtig ist `"green"`.
2. `t.right(winkel)` ist nicht eingerückt und gehört deshalb nicht zur Schleife. Die Turtle läuft siebenmal geradeaus (315 Pixel nach oben) und dreht sich erst danach einmal. `t.right(winkel)` muss genauso weit eingerückt sein wie `t.forward(seite)`.

k16_loesung3.py

```
 1 import turtle
 2
 3
 4 def vieleck(t, ecken, seite):
 5     winkel = 360 / ecken
 6     for i in range(ecken):
 7         t.forward(seite)
 8         t.right(winkel)
 9
10
11 t = turtle.Turtle()
12 t.setPenColor("green")
13 vieleck(t, 7, 45)
```

Aufgabe 4 – Olympische Ringe (`k16_loesung4.py`)

k16_loesung4.py

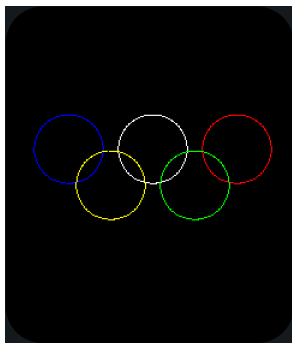
```
 1 import turtle
 2
 3 # Jeder Ring: [x, y, Farbe] (Startpunkt oben am Ring)
 4 RINGE = [
 5     [-70, 50, "blue"],
 6     [0, 50, "white"],
 7     [70, 50, "red"],
 8     [-35, 20, "yellow"],
```

```
    [35, 20, "green"],
]


def ring(t, x, y, farbe):
    t.setPosition(x, y)
    t.setDirection(90)
    t.setPenColor(farbe)
    for i in range(36):         # ein 36-Eck sieht aus wie ein Kreis
        t.forward(5)
        t.right(10)


t = turtle.Turtle()
t.hide()
t.speed(0)
for r in RINGE:
    ring(t, r[0], r[1], r[2])
```

Das 36-Eck hat 36 Seiten von 5 Pixeln; der Umfang ist also 180 Pixel, der Durchmesser etwa 57 Pixel. Die Turtle beginnt oben am Ring, schaut nach rechts und dreht sich 36-mal um 10 Grad. Deshalb ist (x, y) der **oberste** Punkt des Rings, nicht seine Mitte. In der Schleife ist `r` eine Liste wie `[-70, 50, "blue"]`; `r[0]`, `r[1]` und `r[2]` sind x, y und die Farbe.



Aufgabe 5 – Wie weit bis zur Wand? (`k16_loesung5.py`)

k16_loesung5.py

```
import turtle

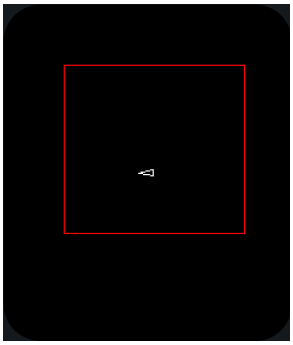

def abstand(t):
    # Zählt die Schritte bis zur roten Wand und fährt dann zurück.
    schritte = 0
    while not t.on_color("red"):
        t.forward(1)
        schritte = schritte + 1
    t.back(schritte)
    return schritte


maurer = turtle.Turtle()
maurer.hide()
maurer.speed(0)
messer = turtle.Turtle()
```

```
19 # ein Zimmer mit roten Wänden
20 maurer.setPenColor("red")
21 maurer.setPosition(-70, -50)
22 for i in range(2):
23     maurer.forward(140)
24     maurer.right(90)
25     maurer.forward(150)
26     maurer.right(90)
27
28 messer.penUp()
29 for richtung in [0, 90, 180, 270]:
30     messer.setDirection(richtung)
31     print(f"Richtung {richtung}: {abstand(messer)} Schritte")
```

Konsole

```
Richtung 0: 87 Schritte
Richtung 90: 77 Schritte
Richtung 180: 47 Schritte
Richtung 270: 67 Schritte
```

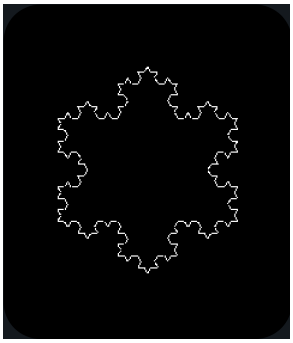


- `abstand` zählt die Schritte, bis Rot vor der Turtle liegt, und schickt die Turtle mit `back(schritte)` an den Start zurück. So beginnt jede Messung in der Mitte.
- Die Wände liegen bei y = 90 (oben), x = 80 (rechts), y = −50 (unten) und x = −70 (links). Jeder Wert ist 3 kleiner als der Abstand, weil `on_color` 3 Pixel vorausschaut.
- Weil der Messer den Stift oben hat (Zeile 28), hinterlässt er keine Spur.

Aufgabe 6 – Schneeflocke (`k16_loesung6.py`)

k16_loesung6.py

```
 1 import turtle
 2
 3
 4 def koch(t, laenge, stufe):
 5     if stufe == 0:
 6         t.forward(laenge)
 7         return
 8     teil = laenge / 3
 9     koch(t, teil, stufe - 1)
10     t.left(60)
11     koch(t, teil, stufe - 1)
12     t.right(120)
13     koch(t, teil, stufe - 1)
14     t.left(60)
15     koch(t, teil, stufe - 1)
```



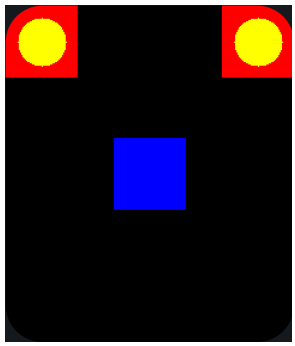
```
18 def schneeflocke(t, laenge, stufe):
19     for i in range(3):
20         koch(t, laenge, stufe)
21         t.right(120)
22
23
24 t = turtle.Turtle()
25 t.hide()
26 t.speed(0)
27 t.setPenColor("white")
28 t.setPosition(-75, 45)
29 t.setDirection(90)
30 schneeflocke(t, 150, 3)
```

Die drei Koch-Kurven bilden ein Dreieck. Wie bei jedem Vieleck dreht sich die Turtle insgesamt einmal herum, nach jeder Kurve also um 360 / 3 = 120 Grad. Weil sie sich nach **rechts** dreht, zeigen die Spitzen der Koch-Kurven (die mit `left(60)` beginnen) nach außen. Mit `left(120)` zeigten sie nach innen.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 17: Der Canvas – flimmerfrei zeichnen

Aufgabe 1 – Was erscheint?



Drei Quadrate, je 60 × 60 Pixel groß: links oben und rechts oben je ein rotes Quadrat mit gelbem Kreis, in der Mitte ein blaues Quadrat ohne Kreis.

`pbc.show` kopiert den Canvas so, wie er **in diesem Moment** ist. Die beiden oberen Kopien bleiben rot, auch wenn der Canvas danach in Zeile 9 blau gefüllt wird. `kachel.fill(pbc.BLUE)` übermalt auch den gelben Kreis, deshalb ist das mittlere Quadrat ganz blau. Rechts oben passt das Quadrat genau: 180 + 60 = 240.

Aufgabe 2 – Finde die Fehler (`k17_loesung2.py`)

1. `flagge.clear(pbc.BLACK)` (Zeile 5) erzeugt einen `AttributeError`: Ein Canvas hat kein `clear`. Richtig ist `flagge.fill(pbc.BLACK)`.
2. Zeile 6 zeichnet mit `pbc.fill_rect` direkt auf den Bildschirm statt in den Canvas. Der rote Streifen fehlt dann in der Flagge. Richtig ist `flagge.fill_rect`.
3. Es fehlt `pbc.show(flagge, 0, 65)`. Ohne `show` bleibt die Flagge im Speicher und ist nie zu sehen. Mit y = 65 steht die 150 Pixel hohe Flagge in der Mitte: (280 − 150) / 2 = 65.

k17_loesung2.py

```
1 import pbc
2
3 pbc.clear(pbc.GRAY)
4 flagge = pbc.Canvas(240, 150)
5 flagge.fill(pbc.BLACK)
6 flagge.fill_rect(0, 50, 240, 50, pbc.RED)
7 flagge.fill_rect(0, 100, 240, 50, pbc.rgb(255, 200, 0))
8 pbc.show(flagge, 0, 65)
```



Aufgabe 3 – Gewitter (`k17_loesung3.py`)

Neu sind der vierte Eintrag in `TAGE` und ein weiterer `elif`-Zweig in `zeichne_tag`:



k17_loesung3.py

```
4 # Tag, Wetter, hoechste und tiefste Temperatur
5 TAGE = [["Montag", "Sonne", 24, 13],
6         ["Dienstag", "Wolken", 19, 11],
7         ["Mittwoch", "Regen", 15, 9],
8         ["Donnerstag", "Gewitter", 22, 16]]
```

k17_loesung3.py

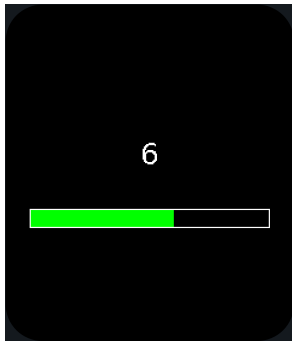
```
29     elif wetter == "Gewitter":
30         # Ab und zu wird der Himmel kurz hell.
31         if schritt % 15 == 0:
32             bild.fill(pbc.WHITE)
33         else:
34             bild.fill(pbc.rgb(60, 60, 80))
35         zeichne_wolke(120, 90)
36         bild.line(125, 110, 110, 140, pbc.YELLOW)
37         bild.line(110, 140, 128, 140, pbc.YELLOW)
38         bild.line(128, 140, 112, 175, pbc.YELLOW)
```

`schritt % 15 == 0` ist nur in jedem 15. Schritt wahr. Dann wird der Himmel für ein einziges Bild weiß, das sieht aus wie ein Blitz. Weil die Liste jetzt vier Tage hat, muss an der Hauptschleife nichts geändert werden: `len(TAGE)` ist automatisch 4.

Aufgabe 4 – Countdown (`k17_loesung4.py`)

k17_loesung4.py

```
 1 import pbc
 2 import time
 3
 4 bild = pbc.Canvas(240, 280)
 5
 6 for zahl in range(10, -1, -1):
 7     bild.fill(pbc.BLACK)
 8     text = str(zahl)
 9     x = (240 - bild.text24_width(text)) // 2
10     bild.text24(text, x, 110, pbc.WHITE)
11     # Der Balken wird mit jeder Sekunde 20 Pixel kuerzer.
12     bild.fill_rect(20, 170, zahl * 20, 16, pbc.GREEN)
13     bild.rect(20, 170, 200, 16, pbc.WHITE)
14     pbc.show(bild)
15     time.sleep(1)
16
17 bild.fill(pbc.BLACK)
18 bild.text24("Start!", 85, 120, pbc.YELLOW)
19 pbc.show(bild)
```

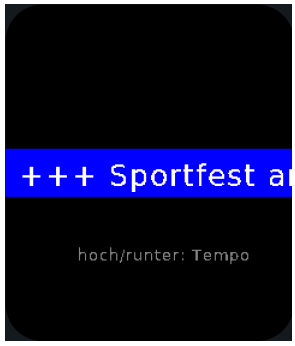


- `range(10, -1, -1)` zählt von 10 bis 0 rückwärts.
- Zeile 9 zentriert die Zahl: Von 240 wird die Breite des Texts abgezogen, der Rest wird halbiert und links als Abstand benutzt.
- Der Balken ist `zahl * 20` Pixel breit: bei 10 also 200 Pixel, bei 0 gar nicht zu sehen. Der weiße Rahmen in Zeile 13 zeigt, wie lang er am Anfang war.

Aufgabe 5 – Laufschrift (`k17_loesung5.py`)

k17_loesung5.py

```
 1 import pbc
 2 import time
 3
 4 TEXT = "+++ Sportfest am Freitag ab 9 Uhr +++"
 5
 6 band = pbc.Canvas(240, 40)
 7 breite = band.text24_width(TEXT)
 8 x = 240
 9 tempo = 3
10
11 pbc.clear()
12 pbc.text12("hoch/runter: Tempo", 60, 200, pbc.GRAY)
13 while True:
14     if pbc.was_pressed_up() and tempo < 10:
15         tempo = tempo + 1
16     if pbc.was_pressed_down() and tempo > 1:
17         tempo = tempo - 1
18     band.fill(pbc.BLUE)
```



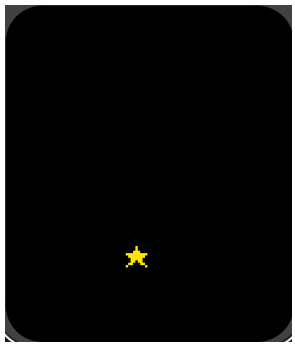
```
19      band.text24(TEXT, x, 8, pbc.WHITE)
20      pbc.show(band, 0, 120)
21      x = x - tempo
22      # Ganz links verschwunden? Dann rechts neu anfangen.
23      if x < -breite:
24          x = 240
25      time.sleep_ms(20)
```

- Der Canvas `band` ist nur 40 Pixel hoch. `pbc.show(band, 0, 120)` erneuert deshalb nur diesen Streifen, der Hinweis aus Zeile 12 bleibt stehen.
- `breite` wird einmal vor der Schleife ausgerechnet. Ist `x` kleiner als `-breite`, ist der Text links ganz hinausgelaufen, und er beginnt wieder bei x = 240, also gerade rechts außerhalb.
- `tempo` ist die Zahl der Pixel pro Schritt. Die Bedingungen `tempo < 10` und `tempo > 1` halten es zwischen 1 und 10.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 18: Sprites – Figuren, die sich bewegen

Aufgabe 1 – Was passiert wann?



Etwa zwei Sekunden lang ist **nichts** zu sehen: Das Sprite ist noch unsichtbar. In Zeile 6 wartet das Programm eine Sekunde, in Zeile 7 gleitet der unsichtbare Stern eine Sekunde lang 100 Pixel nach unten. Erst danach macht Zeile 8 ihn sichtbar, und zwar gleich an der Endposition. Im Terminal steht:

```
Terminal
100 200
```

Auch ein unsichtbares Sprite kann bewegt werden. Man sieht die Bewegung nur nicht.

Aufgabe 2 – Finde die Fehler (`k18_loesung2.py`)

Zeile 4: Der Dateiname braucht die Endung `.png`. Sonst bricht das Programm ab:

```
Terminal
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
OSError: [Errno 2] ENOENT
```

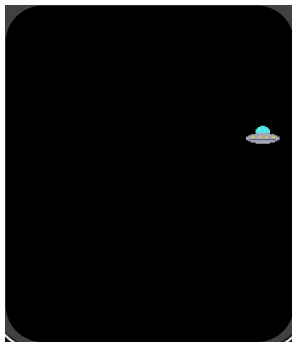
Die beiden anderen Fehler erzeugen keine Meldung:

1. Es fehlt `ufo.show()`. Ohne `show` bleibt das Ufo unsichtbar.
2. `ufo.x = ufo.x + 5` ändert nur die Zahl, das Ufo wird aber nicht neu gezeichnet. Richtig ist `ufo.set_by(5, 0)`.

k18_loesung2.py

```
import pbc
import time

ufo = pbc.sprite("ufo.png")
ufo.set_to(0, 100)
ufo.show()
for i in range(40):
    ufo.set_by(5, 0)
    time.sleep_ms(30)
```



40 Schritte zu je 5 Pixeln: Am Ende steht das Ufo bei x = 200.

Aufgabe 3 – Luftblasen (`k18_loesung3.py`)

Vor der Schleife werden drei Blasen angelegt:

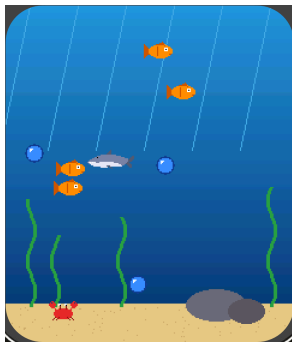
k18_loesung3.py

```
# Luftblasen steigen vom Boden auf.
blasen = []
for i in range(3):
    blase = pbc.sprite("kugel.png")
    blase.set_to(random.randint(0, 220), random.randint(100, 260))
    blase.show()
    blasen.append(blase)
```

In der Schleife, nach der Krabbe:

k18_loesung3.py

```
    for blase in blasen:
        blase.set_by(0, -2)
        # Oben hinaus? Dann unten an neuer Stelle wieder
          anfangen.
        if blase.y < -blase.height:
            blase.set_to(random.randint(0, 220),
              pbc.HEIGHT)
    time.sleep_ms(30)
```



`set_by(0, -2)` bewegt nach **oben**, weil y nach oben kleiner wird. Eine Blase ist ganz oben hinaus, wenn ihre Unterkante über dem Rand liegt, also wenn `blase.y < -blase.height`. Dann beginnt sie bei y = 280, gerade unter dem Bildschirm. Weil die Blasen zuletzt gezeigt wurden, liegen sie über den Fischen.

Aufgabe 4 – Rundflug (`k18_loesung4.py`)

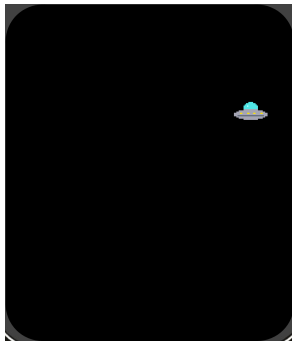
k18_loesung4.py

```
import pbc

# Die Eckpunkte des Rundflugs
PUNKTE = [[20, 30], [190, 30], [190, 230], [20, 230]]

ufo = pbc.sprite("ufo.png")
ufo.set_to(PUNKTE[0][0], PUNKTE[0][1])
ufo.show()

for runde in range(2):
    for punkt in PUNKTE:
        ufo.glide_to(punkt[0], punkt[1], 700)
    # Zurueck zum Anfang
    ufo.glide_to(PUNKTE[0][0], PUNKTE[0][1], 700)
    print("Runde", runde + 1, "fertig")
```



Terminal

```
Runde 1 fertig
Runde 2 fertig
```

Jeder Punkt ist eine Liste `[x, y]`, deshalb heißt die x-Koordinate `punkt[0]` und die y-Koordinate `punkt[1]`. Weil `glide_to` wartet, fliegt das Ufo die Ecken nacheinander an. Der erste `glide_to` in jeder Runde führt zum Startpunkt, an dem das Ufo schon steht; er dauert trotzdem 700 ms.

Aufgabe 5 – Rakete mit Tank (`k18_loesung5.py`)

k18_loesung5.py

```
import pbc
import time

weltraum = pbc.load_image("weltraum.png")
pbc.Sprite._background = weltraum

rakete = pbc.sprite("rakete.png")
rakete.set_to(112, 200)
rakete.show()
tank = 100


def zeige_tank():
    weltraum.fill_rect(10, 10, 104, 14, pbc.WHITE)
    weltraum.fill_rect(12, 12, 100, 10, pbc.BLACK)
    weltraum.fill_rect(12, 12, tank, 10, pbc.GREEN)
    pbc.show(weltraum)


zeige_tank()
```

```
21 while tank > 0:
22     tempo = 2
23     if pbc.pressed_a():
24         tempo = 6
25     dx = 0
26     dy = 0
27     if pbc.pressed_left() and rakete.x >= tempo:
28         dx = -tempo
29     if pbc.pressed_right() and rakete.x + rakete.width + tempo <= pbc.WIDTH:
30         dx = tempo
31     if pbc.pressed_up() and rakete.y >= tempo:
32         dy = -tempo
33     if pbc.pressed_down() and rakete.y + rakete.height + tempo <= pbc.HEIGHT:
34         dy = tempo
35     if dx != 0 or dy != 0:
36         rakete.set_by(dx, dy)
37         # Turbo verbraucht dreimal so viel.
38         tank = max(0, tank - tempo // 2)
39         zeige_tank()
40     time.sleep_ms(20)
41
42 weltraum.text24("Tank leer!", 65, 120, pbc.RED)
43 pbc.show(weltraum)
```



- `zeige_tank` malt den Balken in den Hintergrund-Canvas: einen weißen Rahmen, darin ein schwarzes Feld für den leeren Tank und darüber den grünen Teil, der `tank` Pixel breit ist. `pbc.show(weltraum)` zeigt den neuen Hintergrund mit der Rakete darauf.
- `tempo` ist 2 oder mit Turbo 6. Ein Schritt verbraucht `tempo // 2`, also 1 oder 3. `max(0, …)` verhindert, dass der Tank unter 0 sinkt.
- Der Tank wird nur kleiner, wenn sich die Rakete wirklich bewegt (Zeile 35). Die Schleife läuft, solange `tank > 0` ist. Danach erscheint „Tank leer!“ im Hintergrund.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 19: Kollisionen

Aufgabe 1 – Berührt oder nicht?

Terminal

```
True
False
True
False False
False
```

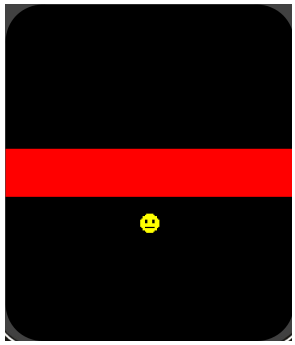
- Ufo bei (0, 0), Stern bei (24, 12): Das Ufo reicht waagrecht bis 28 und senkrecht bis 16. Der Stern beginnt bei 24 und 12, also überlappen sich die Rechtecke: `True`. Das ist die Lage aus der Abbildung in 19.1.
- Stern bei (28, 12): Er beginnt genau dort, wo das Ufo endet: `False`.
- Stern bei (10, 15): Senkrecht überlappt nur die Zeile 15, waagrecht 10 bis 28: `True`. Dabei ist es egal, ob man `stern.touches(ufo)` oder `ufo.touches(stern)` fragt.
- Beide liegen vollständig im Bildschirm: `False False`.
- Ufo bei (212, 100): 212 + 28 = 240. Der rechte Rand liegt genau am Bildschirmrand, ragt aber nicht hinaus: `False`. Erst bei x = 213 wäre es `True`.

Aufgabe 2 – Finde den Fehler (`k19_loesung2.py`)

Die Lava wird mit `pbc.fill_rect` direkt auf den Bildschirm gezeichnet. Sobald sich der Held bewegt, zeichnet der PicoBoy den Bildschirm aus Hintergrund und Sprites neu, und die Lava ist weg. Außerdem gibt es keinen Hintergrund-Canvas, deshalb liefert `on_color` immer `False`. Die Lava muss in einen Canvas, der Hintergrund der Sprites ist (Zeile 4 bis 8):

k19_loesung2.py

```
 1 import pbc
 2 import time
 3
 4 feld = pbc.Canvas(240, 280)
 5 feld.fill(pbc.BLACK)
 6 feld.fill_rect(0, 120, 240, 40, pbc.RED)
 7 pbc.Sprite._background = feld
 8 pbc.show(feld)
 9
10 held = pbc.sprite("held.png")
11 held.set_to(112, 250)
12 held.show()
13
14 while True:
15     if pbc.pressed_up():
16         held.set_by(0, -2)
17     if held.on_color(pbc.RED):
18         print("Lava!")
19         held.set_to(112, 250)
20     time.sleep_ms(20)
```



Terminal

```
Lava!
Lava!
```

Aufgabe 3 – Drei Leben (`k19_loesung3.py`)

Neu ist die Variable `leben = 3`, eine zweite Zeile in `zeige_punkte`:



k19_loesung3.py

```
22 def zeige_punkte():
23     feld.fill(pbc.rgb(0, 70, 40))
24     feld.text18(f"Punkte: {punkte}", 10, 8, pbc.WHITE)
25     feld.text18(f"Leben: {leben}", 140, 8, pbc.WHITE)
26     pbc.show(feld)
```

und ein anderer Umgang mit dem Ball, der unten hinausfällt:

k19_loesung3.py

```
43     ball.set_by(0, dy)
44     if ball.y + ball.height > pbc.HEIGHT:
45         leben = leben - 1
46         zeige_punkte()
47         if leben == 0:
48             break
49         pbc.tone(220, 300)
50         time.sleep_ms(500)
51         ball.set_to(random.randint(20, 200), 40)
52         dy = 3
```

Statt sofort `break` gibt es erst ein Leben weniger. Nur wenn keines mehr übrig ist, endet die Schleife. Sonst startet der Ball oben an einer zufälligen Stelle und fliegt wieder nach unten (`dy = 3`). Die kurze Pause gibt dem Spieler Zeit, sich darauf einzustellen. `zeige_punkte` darf die Variable `leben` lesen, ohne sie zu ändern; deshalb braucht sie kein `global`.

Aufgabe 4 – Punkt im Rechteck (`k19_loesung4.py`)

k19_loesung4.py

```
 1 def punkt_im_rechteck(px, py, x, y, b, h):
 2     return x <= px < x + b and y <= py < y + h
 3
 4
 5 # Rechteck: links oben (10, 20), 30 breit, 15 hoch
 6 print(punkt_im_rechteck(10, 20, 10, 20, 30, 15))  # linke obere Ecke: True
 7 print(punkt_im_rechteck(25, 30, 10, 20, 30, 15))  # mitten drin: True
 8 print(punkt_im_rechteck(40, 30, 10, 20, 30, 15))  # gleich rechts daneben: False
 9 print(punkt_im_rechteck(25, 35, 10, 20, 30, 15))  # gleich darunter: False
10 print(punkt_im_rechteck(5, 5, 10, 20, 30, 15))    # weit weg: False
```

Terminal

```
True
True
False
False
False
```

`x <= px < x + b` ist eine **verkettete** Bedingung: Sie bedeutet dasselbe wie `x <= px and px < x + b`. Das Rechteck mit x = 10 und der Breite 30 umfasst die Pixel 10 bis 39. Bei 40 liegt der Punkt schon daneben, deshalb steht rechts `<` und links `<=`.

Aufgabe 5 – Meteoritenregen (`k19_loesung5.py`)

k19_loesung5.py

```
import pbc
import random
import time

weltraum = pbc.load_image("weltraum.png")
pbc.Sprite._background = weltraum
pbc.show(weltraum)

schiff = pbc.sprite("raumschiff.png")
schiff.set_to(110, 245)
schiff.show()

meteore = []
for i in range(3):
    meteor = pbc.sprite("meteor.png")
    meteor.set_to(random.randint(0, 220), -20 - i * 100)
    meteor.show()
    meteore.append(meteor)

start = time.ticks_ms()
getroffen = False
while not getroffen:
    if pbc.pressed_left() and schiff.x >= 4:
        schiff.set_by(-4, 0)
    if pbc.pressed_right() and schiff.x + schiff.width + 4 <= pbc.WIDTH:
        schiff.set_by(4, 0)
    for meteor in meteore:
        meteor.set_by(0, 4)
        if meteor.y > pbc.HEIGHT:
            meteor.set_to(random.randint(0, 220), -20)
        if meteor.touches(schiff):
            getroffen = True
    time.sleep_ms(10)

sekunden = time.ticks_diff(time.ticks_ms(), start) // 1000
weltraum.text24("Getroffen!", 60, 110, pbc.RED)
weltraum.text18(f"{sekunden} Sekunden", 70, 145, pbc.WHITE)
pbc.show(weltraum)
print("Ueberlebt:", sekunden, "Sekunden")
```

- Die drei Meteore starten über dem Bildschirm in verschiedenen Höhen (Zeile 16: −20, −120, −220). So kommen sie nacheinander herunter.
- Die Variable `getroffen` steuert die Schleife. Weil in Zeile 32 nur `getroffen = True` gesetzt wird, bewegt die `for`-Schleife die restlichen Meteore noch fertig, danach endet die `while`-Schleife.
- `time.ticks_diff` liefert Millisekunden, `// 1000` macht ganze Sekunden daraus.



ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 20: Der Bewegungssensor

Die Programme laufen alle im Emulator. Dort liegt das Gerät aber immer flach, deshalb siehst du die eigentliche Wirkung nur auf dem PicoBoy.

Aufgabe 1 – Wie liegt das Gerät?

Terminal (Emulator)
```
Bildschirm zeigt nach oben
```

Im Emulator ist z immer 9,81, also größer als 8. Liegt der PicoBoy mit dem Bildschirm nach unten, zeigt die z-Achse nach unten: z ist ungefähr −9,8, und das Programm meldet „Bildschirm zeigt nach unten“. Steht das Gerät aufrecht, zeigt die z-Achse zur Seite: z ist ungefähr 0, und es erscheint die letzte Meldung.

Aufgabe 2 – Kipp-Ampel (`k20_loesung2.py`)

k20_loesung2.py
```
 1 import pbc
 2 import time
 3
 4 GRENZE = 2      # m/s²
 5
 6 while True:
 7     x = pbc.accel_x()
 8     if x > GRENZE:          # linker Rand tiefer
 9         pbc.rgb_led(0, 0, 255)
10     elif x < -GRENZE:       # rechter Rand tiefer
11         pbc.rgb_led(255, 0, 0)
12     else:
13         pbc.rgb_led(0, 255, 0)
14     time.sleep_ms(100)
```

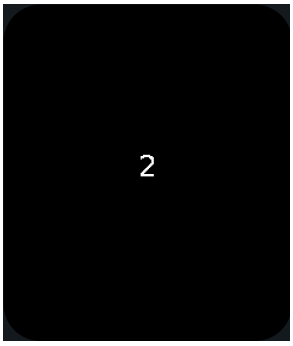
Ist der linke Rand tiefer, zeigt die x-Achse (nach rechts) schräg nach oben: x wird positiv. Ist der rechte Rand tiefer, wird x negativ. Im Emulator leuchtet die LED immer grün.

Aufgabe 3 – Schüttelwürfel (`k20_loesung3.py`)

k20_loesung3.py

```
1  import pbc
2  import random
3  import time
4
5  SCHWELLE = 8    # m/s²: so stark muss sich ein Wert ändern
6
7  def wuerfeln():
8      augen = random.randint(1, 6)
9      pbc.clear()
10     pbc.text24(str(augen), 112, 120, pbc.WHITE)
11     pbc.tone(880, 80)
12
13 pbc.clear()
14 pbc.text18("Schuetteln oder A", 30, 125, pbc.WHITE)
15 alt_x = pbc.accel_x()
16 alt_y = pbc.accel_y()
17 alt_z = pbc.accel_z()
18
19 while True:
20     x = pbc.accel_x()
21     y = pbc.accel_y()
22     z = pbc.accel_z()
23     aenderung = abs(x - alt_x) + abs(y - alt_y) + abs(z - alt_z)
24     if aenderung > SCHWELLE or pbc.was_pressed_a():
25         wuerfeln()
26         time.sleep_ms(500)      # nicht mehrfach pro Schütteln
27     alt_x = x
28     alt_y = y
29     alt_z = z
30     time.sleep_ms(30)
```

- Zeile 23: `aenderung` addiert, wie stark sich jede Achse seit der letzten Messung verändert hat. `abs` macht negative Unterschiede positiv. Liegt das Gerät ruhig, ist die Summe klein; beim Schütteln wird sie groß.
- Zeile 24: Taste A würfelt auch, damit man im Emulator testen kann (Bild: Wurf nach einem Druck auf A).
- Zeile 26: Nach einem Wurf wartet das Programm eine halbe Sekunde. Sonst würde ein einziges Schütteln mehrmals hintereinander würfeln.



Ist der Würfel auf deinem Gerät zu empfindlich oder zu träge, verändere `SCHWELLE`.

Aufgabe 4 – Mittelwert der letzten fünf (`k20_loesung4.py`)

k20_loesung4.py

```
import pbc
import random

ANZAHL = 5
letzte = []

for i in range(12):
    # künstliches Zittern wie in Abschnitt 20.4
    roh = pbc.accel_x() + random.randint(-20, 20) / 10
    letzte.append(roh)
    if len(letzte) > ANZAHL:
        letzte.pop(0)           # ältesten Wert entfernen
    mittel = sum(letzte) / len(letzte)
    print(f"roh: {roh:5.1f}   Mittel aus {len(letzte)}: {mittel:5.2f}")
```

Terminal (Emulator, zufällig)

```
roh:   0.7   Mittel aus 1:  0.70
roh:   1.5   Mittel aus 2:  1.10
roh:  -0.2   Mittel aus 3:  0.67
roh:   1.6   Mittel aus 4:  0.90
roh:   1.6   Mittel aus 5:  1.04
roh:  -1.6   Mittel aus 5:  0.58
roh:   0.0   Mittel aus 5:  0.28
roh:  -0.5   Mittel aus 5:  0.22
roh:  -1.6   Mittel aus 5: -0.42
roh:   2.0   Mittel aus 5: -0.34
roh:  -2.0   Mittel aus 5: -0.42
roh:   1.7   Mittel aus 5: -0.08
```

Die Liste `letzte` wächst bis auf fünf Werte. Danach entfernt `pop(0)` bei jedem neuen Wert den ältesten (Index 0). Am Anfang stehen weniger als fünf Werte in der Liste; darum teilen wir durch `len(letzte)` und nicht durch 5.

Aufgabe 5 – Münzen einsammeln (`k20_loesung5.py`)

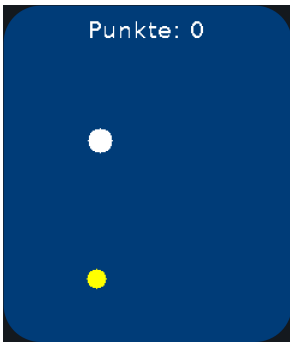
k20_loesung5.py

```
import pbc
import random
import time

bild = pbc.Canvas(240, 280)
R_KUGEL = 10
R_MUENZE = 8
x = 120
y = 140
mx = random.randint(20, 220)
my = random.randint(40, 260)
punkte = 0
```

```
14 while True:
15     # Kippen (Gerät) und Joystick (Emulator) bewegen die Kugel
16     x = x - pbc.accel_x() * 0.8
17     y = y - pbc.accel_y() * 0.8
18     if pbc.pressed_left():
19         x -= 4
20     if pbc.pressed_right():
21         x += 4
22     if pbc.pressed_up():
23         y -= 4
24     if pbc.pressed_down():
25         y += 4
26     x = max(R_KUGEL, min(239 - R_KUGEL, x))
27     y = max(R_KUGEL, min(279 - R_KUGEL, y))
28
29     # Abstand wie in Kapitel 19 - ohne Wurzel, mit Quadraten
30     dx = x - mx
31     dy = y - my
32     if dx * dx + dy * dy < (R_KUGEL + R_MUENZE) ** 2:
33         punkte += 1
34         pbc.tone(1200, 60)
35         mx = random.randint(20, 220)
36         my = random.randint(40, 260)
37
38     bild.fill(pbc.rgb(0, 60, 120))
39     bild.ellipse(mx, my, R_MUENZE, R_MUENZE, pbc.YELLOW, True)
40     bild.ellipse(int(x), int(y), R_KUGEL, R_KUGEL, pbc.WHITE, True)
41     bild.text18(f"Punkte: {punkte}", 70, 10, pbc.WHITE)
42     pbc.show(bild)
43     time.sleep_ms(30)
```

- Zeile 16–17: Kippen bewegt die Kugel wie in 20.3.
- Zeile 18–25: Der Joystick bewegt sie zusätzlich um 4 Pixel. Im Emulator ist das die einzige Steuerung (Bild: nach links und oben gefahren).
- Zeile 30–32: Die Kugel berührt die Münze, wenn der Abstand der Mittelpunkte kleiner ist als die Summe der Radien. Statt die Wurzel zu ziehen, vergleichen wir die Quadrate: `dx * dx + dy * dy` mit `(R_KUGEL + R_MUENZE) ** 2`.



ANHANG C – LÖSUNGEN

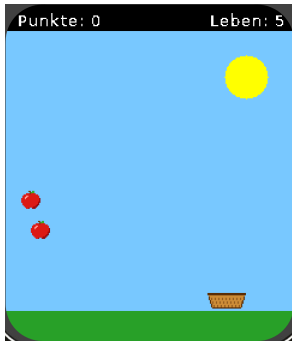
Lösungen zu Kapitel 21: Projekt: Ein Spiel Schritt für Schritt

Aufgabe 1 – Was passiert?

1. Der Korb bleibt stehen. Zeile 24 bewegt ihn nach links nur, wenn `korb.x - TEMPO_KORB >= 0` ist, er also nach dem Schritt noch ganz im Bild wäre. Weil der Korb bei x = 104 startet und 12 Pixel pro Schritt fährt, hält er bei x = 8 an. Ein weiterer Schritt würde x = −4 ergeben.

2. Die Äpfel sollen von **oberhalb** des Bildschirms hereinfallen und nicht plötzlich mitten im Bild auftauchen. Weil jeder Apfel ein anderes negatives y bekommt (−150 bis −20), kommen sie nacheinander an und nicht alle gleichzeitig.
3. Das ganze Spiel wird deutlich langsamer und ruckelt. Jedes `set_by` zeichnet den kompletten Bildschirm neu (im Emulator etwa 40 ms). Mit 8 Äpfeln sind das pro Durchlauf der Spielschleife 8 Neuzeichnungen statt 3, also mehr als doppelt so lange. Äpfel **und** Korb bewegen sich deshalb langsamer.

Aufgabe 2 – Ein leichteres Spiel (`k21_loesung2.py`)



Geändert sind nur die Konstanten oben und die Zeile, in der `leben` seinen Startwert bekommt:

k21_loesung2.py

```
 7 HIMMEL = pbc.rgb(120, 200, 255)
 8 WIESE = pbc.rgb(40, 160, 40)
 9 KORB_Y = 240
10 TEMPO_KORB = 16
11 TEMPO_APFEL = 8
12 ANZAHL_AEPFEL = 2
13 LEBEN_START = 5
14 DATEI = "apfel_rekord.txt"
```

```
leben = LEBEN_START
```

Zeile 10 (`TEMPO_KORB = 16`), Zeile 12 (`ANZAHL_AEPFEL = 2`), Zeile 13 ist neu, und in Zeile 83 steht `leben = LEBEN_START` statt `leben = 3`. Weil alle wichtigen Zahlen als Konstanten oben stehen, musste im Rest des Programms nichts geändert werden.

Aufgabe 3 – Finde die Fehler (`k21_loesung3.py`)



Beim Spielen von `k21_aufgabe3.py` fällt auf: Ein gefangener Apfel zählt mehrere Punkte, und am Ende steht „Leben: -1“ (Bild rechts).

1. **Zeile 65 bis 68:** Nach einem Fang fehlt `nach_oben(apfel)`. Der Apfel fällt deshalb weiter durch den Korb und berührt ihn in mehreren Durchläufen hintereinander. Jedes Mal gibt es einen Punkt. Danach fällt er ins Gras und kostet sogar noch ein Leben.
2. **Zeile 61:** `while leben >= 0` läuft auch bei 0 Leben weiter. Das Spiel endet erst, wenn `leben` −1 ist. Richtig ist `while leben > 0`.

Die korrigierte Fassung ist dieselbe wie Stufe 3:

k21_loesung3.py

```
60 # Spielschleife
61 while leben > 0:
62     bewege_korb(korb)
```

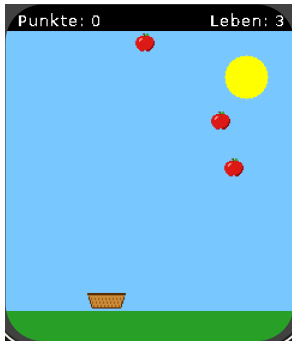
```
63      for apfel in aepfel:
64          apfel.set_by(0, TEMPO_APFEL)
65          if apfel.touches(korb):
66              punkte += 1
67              pbc.tone(880, 50)
68              nach_oben(apfel)
69              zeige_stand(feld, punkte, leben)
70          elif apfel.y > 260:
71              leben -= 1
72              pbc.tone(220, 200)
73              nach_oben(apfel)
74              zeige_stand(feld, punkte, leben)
75              if leben == 0:
76                  break
77      time.sleep_ms(20)
```

Solche Fehler heißen **Logikfehler**: Python meldet nichts, weil jede Zeile für sich erlaubt ist. Man findet sie nur durch genaues Testen.

Aufgabe 4 – Immer schneller (`k21_loesung4.py`)

k21_loesung4.py

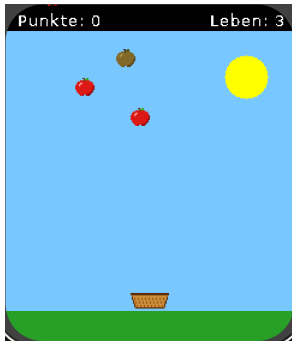
```
85 # Spielschleife
86 while leben > 0:
87     # alle 5 Punkte fallen die Aepfel 1 Pixel schneller
88     tempo = TEMPO_APFEL + punkte // 5
89     bewege_korb(korb)
90     for apfel in aepfel:
91         apfel.set_by(0, tempo)
```



`punkte // 5` ist die ganzzahlige Division: Bei 0 bis 4 Punkten ergibt sie 0, bei 5 bis 9 Punkten 1 usw. Zu `TEMPO_APFEL = 8` kommt also alle 5 Punkte ein Pixel dazu. Die Variable `tempo` wird am Anfang jedes Durchlaufs neu berechnet und in Zeile 91 statt `TEMPO_APFEL` benutzt.

Aufgabe 5 – Der faule Apfel (`k21_loesung5.py`)

Das Bild `faulapfel.png` muss zuerst hochgeladen werden (**Dateien → Bilder verwalten…**). Der faule Apfel ist ein eigenes Sprite, das nicht in die Liste kommt, weil für ihn andere Regeln gelten:



k21_loesung5.py

```
81 faul = pbc.sprite("faulapfel.png")
82 nach_oben(faul)
83 faul.show()
84
85 punkte = 0
86 leben = 3
87 zeige_stand(feld, punkte, leben)
```

```
 89 # Spielschleife
 90 while leben > 0:
 91     bewege_korb(korb)
 92     faul.set_by(0, TEMPO_APFEL)
 93     if faul.touches(korb):
 94         leben -= 1
 95         pbc.tone(150, 300)
 96         nach_oben(faul)
 97         zeige_stand(feld, punkte, leben)
 98         if leben == 0:
 99             break
100     elif faul.y > 260:
101         nach_oben(faul)
```

- Zeile 81 bis 83 legen den faulen Apfel an, genau wie einen roten.
- Zeile 93 bis 99: Fängt der Korb ihn, kostet das ein Leben (mit einem besonders tiefen Ton). Ist es das letzte Leben, beendet `break` in Zeile 99 sofort die `while`-Schleife. Sonst könnte im selben Durchlauf noch ein roter Apfel ins Gras fallen und „Leben: -1“ erzeugen.
- Zeile 100 und 101: Fällt er ins Gras, startet er einfach neu, ohne Strafe.
- Am Spielende wird auch `faul` mit `faul.hide()` versteckt (Zeile 120).

Lösungen zu Kapitel 22: Dein eigenes Projekt

Aufgabe 1 – Ein Plan für die Stoppuhr

1. **Was soll das Programm tun?** Es misst die Zeit zwischen Start und Stopp und zeigt sie in Sekunden mit einer Nachkommastelle an.
2. **Wie bedient man es?** A startet und stoppt, B setzt auf 0 zurück. Auf dem Display steht groß die Zeit, darunter klein „A = Start/Stopp, B = Null“.
3. **Kleinste Fassung:** A startet, ein zweites A stoppt und zeigt die gemessene Zeit.

Stufe	Was kommt dazu?	Fertig, wenn …
1	Mit A Start und Stopp, Zeit mit `time.ticks_ms()` und `time.ticks_diff()` messen, Ergebnis in der Konsole	nach zwei Drücken eine plausible Zeit im Terminal steht
2	Zeit auf dem Display, Sekunden mit einer Nachkommastelle	die Anzeige mit einer Handy-Stoppuhr übereinstimmt
3	Zeit läuft während der Messung sichtbar mit	die Zahl beim Laufen hochzählt, ohne zu flackern
4	Taste B setzt zurück	nach B wieder 0.0 steht

Andere sinnvolle Pläne sind natürlich auch richtig. Wichtig ist, dass jede Stufe für sich testbar ist.

Aufgabe 2 – Fehlersuche mit `print` (`k22_loesung2.py`)

Eine `print`-Zeile am Ende der Schleife zeigt, was in jedem Durchlauf passiert:

k22_loesung2_print.py

```
4 for t in temperaturen:
5     warme_tage = 0
6     if t > 20:
7         warme_tage += 1
8     print("t:", t, "warme_tage:", warme_tage)
```

Terminal

```
t: 18 warme_tage: 0
t: 22 warme_tage: 1
t: 25 warme_tage: 1
t: 19 warme_tage: 0
t: 21 warme_tage: 1
t: 30 warme_tage: 1
t: 17 warme_tage: 0
Warme Tage: 0
```

`warme_tage` wird bei jedem Tag wieder auf 0 gesetzt, weil `warme_tage = 0` **in** der Schleife steht. Es kann also nie größer als 1 werden, und am Ende zählt nur der letzte Tag (17 Grad). Die Zeile muss vor die Schleife:

k22_loesung2.py

```
1 # Wie viele Tage waren waermer als 20 Grad? – korrigiert
2 temperaturen = [18, 22, 25, 19, 21, 30, 17]
3
4 warme_tage = 0
5 for t in temperaturen:
6     if t > 20:
7         warme_tage += 1
8 print("Warme Tage:", warme_tage)
```

Terminal

```
Warme Tage: 4
```

Aufgabe 3 – Verkleinern (`k22_loesung3.py`)

Das verkleinerte Programm braucht weder `pbc` noch die Funktion `balken`:

k22_loesung3_klein.py

```
1 tore = [2, 0, 3, 1, 4]
2 for i in range(len(tore) + 1):
3     print(i, tore[i])
```

Terminal

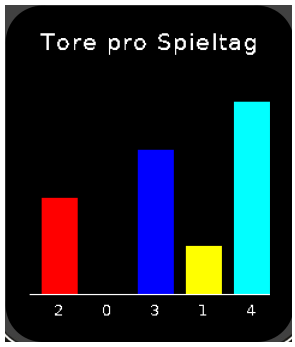
```
0 2
1 0
2 3
3 1
4 4

Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError: list index out of range
```

Die Liste hat 5 Elemente mit den Plätzen 0 bis 4. `range(len(tore) + 1)` läuft aber von 0 bis 5. Beim Platz 5 gibt es kein Element, daher der `IndexError`. Im Diagramm-Programm fällt das nicht auf, weil vorher alle fünf Balken schon gezeichnet sind. Korrigiert wird Zeile 18:

k22_loesung3.py

```
15 pbc.clear()
16 pbc.text18("Tore pro Spieltag", 29, 20, pbc.WHITE)
17 pbc.hline(20, 240, 200, pbc.WHITE)
18 for i in range(len(tore)):
19     balken(i, tore[i], FARBEN[i])
```



`range(len(tore))` liefert genau die Plätze, die es gibt.

Aufgabe 4 – Würfel mit Statistik (`k22_loesung4.py`)

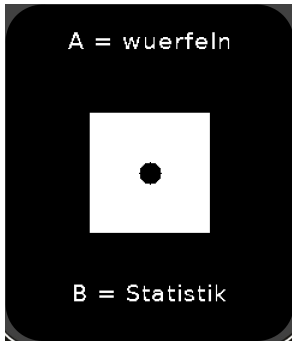
Die Funktion `zeichne_wuerfel` und die Liste `AUGEN` bleiben wie in `k22_schritt3.py`. Neu ist der Teil ab Zeile 26:

k22_loesung4.py

```
26 pbc.clear()
27 pbc.text18("A = wuerfeln", 52, 20, pbc.WHITE)
28 pbc.text18("B = Statistik", 55, 230, pbc.WHITE)
29 haeufigkeit = [0, 0, 0, 0, 0, 0, 0]  # benutzt werden Index 1 bis 6
30 while True:
31     if pbc.was_pressed_a():
32         zahl = random.randint(1, 6)
33         haeufigkeit[zahl] += 1
34         zeichne_wuerfel(zahl)
```

```
35      if pbc.was_pressed_b():
36          print("Statistik nach", sum(haeufigkeit), "Wuerfen:")
37          for zahl in range(1, 7):
38              print(f"  {zahl}: {haeufigkeit[zahl]}-mal")
39      time.sleep_ms(20)
```

- Zeile 29: `haeufigkeit` hat sieben Plätze. Platz 0 wird nie benutzt, so steht die Anzahl der Einsen auf Platz 1, die der Zweien auf Platz 2 usw.
- Zeile 33: Nach jedem Wurf wird der passende Zähler um 1 erhöht.
- Zeile 35 bis 38: Taste B gibt die Statistik aus. `sum(haeufigkeit)` ist die Zahl aller Würfe.



Nach fünf Würfen und einem Druck auf B steht im Terminal zum Beispiel:

```
Terminal
Statistik nach 5 Wuerfen:
  1: 2-mal
  2: 0-mal
  3: 1-mal
  4: 1-mal
  5: 0-mal
  6: 1-mal
```

ANHANG C – LÖSUNGEN

Lösungen zum Zusatzkapitel: Objekte und Klassen

Aufgabe 1 – Zwei Ampeln

```
Terminal
gruen rot-gelb
rot-gelb 1
```

`a` und `b` sind zwei verschiedene Objekte, jedes hat sein eigenes Attribut `nr`. `a` wurde zweimal weitergeschaltet (`nr` = 2, „gruen“), `b` einmal (`nr` = 1, „rot-gelb“). Danach schaltet die Schleife `a` dreimal weiter: 2 → 3 → 0 → 1. Nach „gelb“ (3) kommt wieder „rot“ (0), weil `(3 + 1) % 4` gleich 0 ist. Am Ende ist `a.nr` also 1, die Farbe „rot-gelb“.

Aufgabe 2 – Finde die Fehler (`kZ_loesung2.py`)

Python meldet die Fehler einzeln, nacheinander. Zuerst kommt ein `TypeError` in Zeile 16:

```
Terminal
Traceback (most recent call last):
  File "<stdin>", line 16, in <module>
TypeError: mittel() takes 0 positional arguments but 1 were given
```

Beim Aufruf `hof.mittel()` gibt Python das Objekt als `self` mit, also einen Wert („1 were given“). Die Methode `mittel` in Zeile 9 hat aber gar keinen Parameter („takes 0“). Richtig: `def mittel(self):`. Nach dieser Korrektur kommt ein `NameError` in Zeile 17:

Terminal

```
Schulhof 15.0

Traceback (most recent call last):
  File "<stdin>", line 17, in <module>
NameError: name 'werte' isn't defined
```

Die Liste `werte` ist ein Attribut des Objekts `hof`. Außerhalb der Klasse heißt sie `hof.werte`. Das korrigierte Programm:

kZ_loesung2.py

```
 1 class Thermometer:
 2     def __init__(self, ort):
 3         self.ort = ort
 4         self.werte = []
 5
 6     def messen(self, wert):
 7         self.werte.append(wert)
 8
 9     def mittel(self):
10         return sum(self.werte) / len(self.werte)
11
12 hof = Thermometer("Schulhof")
13 hof.messen(12)
14 hof.messen(15)
15 hof.messen(18)
16 print(hof.ort, hof.mittel())
17 print("Hoechster Wert:", max(hof.werte))
```

Terminal

```
Schulhof 15.0
Hoechster Wert: 18
```

Aufgabe 3 – Rechtecke erweitern (`kZ_loesung3.py`)

kZ_loesung3.py

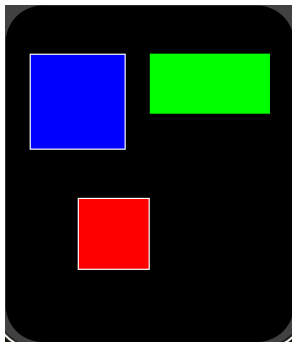
```
1 import pbc
2
3 class Rechteck:
4     def __init__(self, x, y, breite, hoehe, farbe):
5         self.x = x
6         self.y = y
7         self.breite = breite
8         self.hoehe = hoehe
9         self.farbe = farbe
```

```
11      def flaeche(self):
12          return self.breite * self.hoehe
13
14      def umfang(self):
15          return 2 * (self.breite + self.hoehe)
16
17      def ist_quadrat(self):
18          return self.breite == self.hoehe
19
20      def zeichne(self):
21          pbc.fill_rect(self.x, self.y, self.breite, self.hoehe, self.farbe)
22          if self.ist_quadrat():
23              pbc.rect(self.x, self.y, self.breite, self.hoehe, pbc.WHITE)
24
25  pbc.clear()
26  formen = [Rechteck(20, 40, 80, 80, pbc.BLUE),
27            Rechteck(120, 40, 100, 50, pbc.GREEN),
28            Rechteck(60, 160, 60, 60, pbc.RED)]
29  for r in formen:
30      r.zeichne()
31      q = r.ist_quadrat()
32      print(r.breite, "x", r.hoehe, "Umfang:", r.umfang(), "Quadrat:", q)
```

Terminal

```
80 x 80 Umfang: 320 Quadrat: True
100 x 50 Umfang: 300 Quadrat: False
60 x 60 Umfang: 240 Quadrat: True
```

`zeichne` ruft in Zeile 22 die Methode `ist_quadrat` desselben Objekts auf: `self.ist_quadrat()`. Die Schleife am Ende behandelt alle Rechtecke gleich, egal wie viele in der Liste `formen` stehen.



Aufgabe 4 – Das Sparschwein (`kZ_loesung4.py`)

kZ_loesung4.py

```
1  class Konto:
2      def __init__(self, besitzer):
3          self.besitzer = besitzer
4          self.stand = 0
5
6      def einzahlen(self, betrag):
7          self.stand = self.stand + betrag
```

```
    def abheben(self, betrag):
        if betrag > self.stand:
            return False
        self.stand = self.stand - betrag
        return True

    def __str__(self):
        return f"Konto von {self.besitzer}: {self.stand} Euro"

sparschwein = Konto("Jonas")
sparschwein.einzahlen(20)
sparschwein.einzahlen(15)
print(sparschwein)
print("Kino (12 Euro):", sparschwein.abheben(12))
print("Fahrrad (150 Euro):", sparschwein.abheben(150))
print(sparschwein)
```

Terminal

```
Konto von Jonas: 35 Euro
Kino (12 Euro): True
Fahrrad (150 Euro): False
Konto von Jonas: 23 Euro
```

In `abheben` beendet das erste `return False` die Methode sofort, wenn zu wenig Geld da ist. Nur sonst wird der Betrag abgezogen. So kann der Kontostand nie unter 0 fallen.

Aufgabe 5 – Bälle, die sich berühren (`kZ_loesung5.py`)

kZ_loesung5.py

```
import pbc
import random
import time
import math

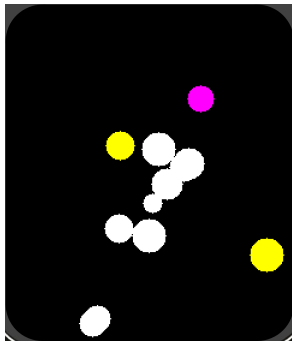
class Ball:
    def __init__(self, farbe):
        self.r = random.randint(8, 14)
        self.x = random.randint(self.r, pbc.WIDTH - self.r)
        self.y = random.randint(self.r, pbc.HEIGHT - self.r)
        self.dx = random.choice([-3, -2, 2, 3])
        self.dy = random.choice([-3, -2, 2, 3])
        self.farbe = farbe

    def bewege(self):
        self.x = self.x + self.dx
        self.y = self.y + self.dy
        if self.x < self.r or self.x > pbc.WIDTH - self.r:
            self.dx = -self.dx
        if self.y < self.r or self.y > pbc.HEIGHT - self.r:
            self.dy = -self.dy
```

```
23      def beruehrt(self, anderer):
24          abst_x = self.x - anderer.x
25          abst_y = self.y - anderer.y
26          abstand = math.sqrt(abst_x ** 2 + abst_y ** 2)
27          return abstand < self.r + anderer.r
28
29      def zeichne(self, bild, farbe):
30          bild.ellipse(self.x, self.y, self.r, self.r, farbe, True)
```

`beruehrt` (Zeilen 23–27) berechnet den Abstand der beiden Mittelpunkte mit dem Satz des Pythagoras und vergleicht ihn mit der Summe der Radien. Die Methode bekommt neben `self` ein zweites `Ball`-Objekt als Parameter und kann auch dessen Attribute lesen (`anderer.x`).

`zeichne` hat jetzt einen Parameter `farbe`, damit das Hauptprogramm entscheiden kann, ob der Ball in seiner eigenen Farbe oder weiß erscheint.

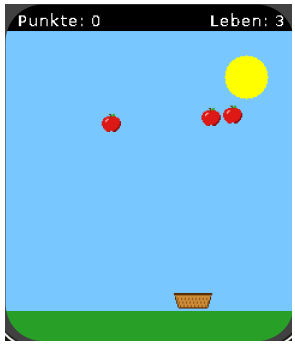


kZ_loesung5.py

```
32 FARBEN = [pbc.RED, pbc.GREEN, pbc.BLUE, pbc.YELLOW, pbc.CYAN, pbc.MAGENTA]
33 bild = pbc.Canvas(pbc.WIDTH, pbc.HEIGHT)
34 baelle = [Ball(f) for f in FARBEN + FARBEN]
35
36 while True:
37     bild.fill(pbc.BLACK)
38     for ball in baelle:
39         ball.bewege()
40     for ball in baelle:
41         farbe = ball.farbe
42         for anderer in baelle:
43             if anderer != ball and ball.beruehrt(anderer):
44                 farbe = pbc.WHITE
45         ball.zeichne(bild, farbe)
46     pbc.show(bild)
47     time.sleep_ms(20)
```

Zuerst werden **alle** Bälle bewegt (Zeilen 38–39), danach wird für jeden Ball geprüft, ob er einen anderen berührt (Zeilen 40–45). Würde man in derselben Schleife bewegen und prüfen, stünden manche Bälle schon an der neuen und andere noch an der alten Stelle. `anderer != ball` sorgt dafür, dass sich ein Ball nicht selbst berührt. `FARBEN + FARBEN` in Zeile 34 ergibt eine Liste mit zwölf Farben, also zwölf Bälle.

Aufgabe 6 – Der faule Apfel mit Klassen (`kZ_loesung6.py`)



Geändert werden nur drei Stellen. Die Klasse `Apfel` bekommt die Parameter `bild` und `faul` und merkt sich `faul` als Attribut:

kZ_loesung6.py

```
29 class Apfel:
30     TEMPO = 8
31
32     def __init__(self, bild, faul=False):
33         self.sprite = pbc.sprite(bild)
34         self.faul = faul
35         self.nach_oben()
36         self.sprite.show()
```

Im Hauptprogramm entstehen drei rote und ein fauler Apfel. In der Spielschleife fragt jeder Apfel sein Attribut `faul` ab:

kZ_loesung6.py

```
108 korb = Korb()
109 aepfel = [Apfel("apfel.png") for i in range(ANZAHL_AEPFEL)]
110 aepfel.append(Apfel("faulapfel.png", True))
111 stand = Spielstand(feld)
112
113 # Spielschleife
114 while not stand.vorbei():
115     korb.bewege()
116     for apfel in aepfel:
117         apfel.falle()
118         if apfel.gefangen(korb):
119             if apfel.faul:
120                 stand.verpasst()
121             else:
122                 stand.treffer()
123             apfel.nach_oben()
124         elif apfel.im_gras():
125             if not apfel.faul:
126                 stand.verpasst()
127             apfel.nach_oben()
128         if stand.vorbei():
129             break
130     time.sleep_ms(20)
```

`Apfel("apfel.png")` benutzt den Standardwert `faul=False`. Alle anderen Methoden, also `falle`, `gefangen`, `im_gras` und `nach_oben`, bleiben unverändert: Ein fauler Apfel fällt genauso wie ein guter. Die Abfrage `if stand.vorbei()` steht jetzt **nach** beiden Fällen (Zeilen 128–129), weil auch ein gefangener fauler Apfel das letzte Leben kosten kann. In Kapitel 21 musste man dafür an mehreren Stellen im Programm Neues einbauen; hier steckt das Wissen „bin ich faul?“ im Apfel selbst.

ANHANG D

Für Fortgeschrittene: Bilder, Dateien und Firmware

Hier steht, wie du eigene Bilder für Sprites und Hintergründe auf den PicoBoy oder in den Emulator bringst, wie du Dateien auf dem Gerät verwaltest, was die Konsole am echten Gerät noch kann und wie MicroPython auf den PicoBoy kommt.

D.1 Wo liegen Bilder und Dateien?

Programme, die Bilder laden (`pbc.load_image`, `pbc.sprite`), suchen die Bilddatei **dort, wo das Programm läuft**:

Ziel	Bilder liegen …	Dateien aus `open()` liegen …
PBC+ (Gerät)	im Speicher des PicoBoy	im Speicher des PicoBoy, dauerhaft
Emulator	im Browser (bleiben erhalten, auch nach dem Neuladen)	im Emulator, bis die Seite neu geladen wird

Die Menüpunkte unter **Dateien → Daten auf dem PBC+** arbeiten immer mit dem Ziel, das oben rechts eingestellt ist. Steht der Schalter auf **Emulator**, heißt der Dialog zum Beispiel „Bilder auf dem Emulator“; steht er auf **PBC+**, muss das Gerät verbunden sein.

D.2 Bilder hochladen: Dateien → Bilder verwalten…

Der Dialog zeigt alle PNG-Bilder des Ziels mit Größe. Mit ⭳ lädst du ein Bild herunter, mit × löschst du es. **PNG hochladen…** öffnet den Dialog „Bild bearbeiten und hochladen“:

1. **Bild von der Festplatte wählen…** – jedes gängige Bildformat geht, es wird als PNG gespeichert.
2. Mit dem blauen Rechteck wählst du einen Ausschnitt; unter **Ausgabegröße** stellst du Breite und Höhe ein (höchstens 240 × 280, die Größe des Displays). Sprites sind meist 16 bis 48 Pixel groß.
3. **Dateiname** eingeben (ohne Umlaute und Leerzeichen, `.png` wird ergänzt) und **Hochladen**. Eine Datei mit gleichem Namen wird ohne Rückfrage ersetzt.

ZIP hochladen… lädt alle PNG-Bilder einer ZIP-Datei unverändert und ohne Dialog hoch.

Durchsichtige Stellen: Pixel, die im PNG durchsichtig sind (weniger als 50 % deckend), bleiben auf dem PicoBoy durchsichtig; alle anderen werden voll deckend gezeichnet. Halb durchsichtige, weichgezeichnete Kanten werden also entweder ganz sichtbar oder ganz unsichtbar und können ausgefranst wirken. Bilder für Sprites zeichnest du deshalb am besten pixelgenau, ohne Kantenglättung.

D.3 Eigene Bilder malen: Dateien → Bild zeichnen…

PBC+ Studio hat einen kleinen Pixel-Editor. Zuerst wählst du die Größe (16 × 16, 32 × 32, 48 × 48, 64 × 64, das ganze Display 240 × 280 oder eigene Werte), dann malst du mit Stift, Linie, Rechteck und Ellipse. ⌫ **Radieren** macht Pixel durchsichtig. **Speichern** fragt nach einem Dateinamen und speichert das Bild auf dem Ziel. Hier gibt es keine halb durchsichtigen Pixel, die Bilder eignen sich also gut für Sprites und für `on_color`.

D.4 Dateien auf dem Gerät: Dateien → Dateien verwalten…

Bei verbundenem PicoBoy zeigt der Dialog „Dateien auf dem PBC+“ alle Dateien im Speicher des Geräts – Programme, Bilder und Dateien, die deine Programme mit `open()` geschrieben haben. Du kannst Dateien herunterladen (⭳), löschen (×) und mit **⬆ Datei(en) oder ZIP hochladen…** beliebige Dateien auf das Gerät kopieren, zum Beispiel eine Textdatei mit Daten für dein Programm.

Eine ZIP-Datei wird entpackt: Alle Dateien landen ohne Unterordner direkt im Speicher, geöffnet wird nichts. `boot.py`, `main.py` und `pbc.py` aus der ZIP werden ausgelassen, ebenso die zweite von zwei Dateien mit gleichem Namen (das Terminal nennt sie). Vorhandene Dateien werden ohne Rückfrage ersetzt. Reicht der Platz auf dem Gerät nicht, wird gar nichts hochgeladen.

Steht das Ziel auf **Emulator**, heißt der Dialog „Dateien auf dem Emulator“ und zeigt die Bilder und Dateien, die du in den Browser hochgeladen hast. Dateien, die ein Programm im Emulator mit `open()` schreibt, erscheinen dort nicht; sie verschwinden beim Neuladen der Seite.

> `boot.py`, `main.py` und `pbc.py` sind als **System** markiert. Lösche sie nicht: `pbc.py` enthält die PicoBoy-Befehle, `main.py` startet beim Einschalten.

Programme auf dem Gerät: Dateien → Auf den PBC+… speichert das Programm aus dem Editor (die Endung `.py` wird ergänzt, vor dem Überschreiben fragt PBC+ Studio nach). **Dateien → Vom PBC+…** lädt ein Programm vom Gerät in den Editor. Beides geht auch mit dem Ziel **Emulator**.

Übertragen: Dateien → PBC+ → Emulator… kopiert Dateien vom verbundenen Gerät in den Emulator, **Emulator → PBC+…** in die andere Richtung, zum Beispiel alle Bilder, die du im Emulator hochgeladen hast.

D.5 Das Startmenü

Dateien → Startmenü installieren… schreibt auf das Gerät eine `main.py` mit diesen zwei Zeilen:

```
import pbc
pbc.menu()
```

`main.py` startet der PicoBoy automatisch beim Einschalten. `pbc.menu()` zeigt eine Liste aller Programme auf dem Gerät; mit dem Joystick wählst du eins aus. Eine vorhandene `main.py` wird vorher in `main(2).py` umbenannt, sie geht also nicht verloren.

D.6 Die REPL am Gerät

Ist der PicoBoy verbunden (Ziel **PBC+**, **Verbinden**), gehen deine Tastendrücke im Terminal direkt an MicroPython auf dem Gerät. Läuft gerade kein Programm, siehst du die Eingabeaufforderung `>>>`. Das ist die **REPL** (Read – Eval – Print – Loop): Du tippst eine Anweisung, drückst **Enter**, und MicroPython führt sie sofort aus.

```
>>> 7 * 6
42
>>> import pbc
>>> pbc.led_green(True)
>>>
```

- Läuft ein Programm, hältst du es mit **Strg + C** an; danach erscheint `>>>`.
- **Soft-Reset** über dem Terminal startet MicroPython auf dem Gerät neu.
- **Konsole leeren** löscht nur die Anzeige im Terminal.

Im Emulator gibt es keine REPL; dort reagiert das Terminal nur, wenn ein Programm mit `input()` auf eine Eingabe wartet.

D.7 MicroPython auf den PicoBoy bringen

Normalerweise ist MicroPython schon auf dem PicoBoy. Wenn nicht – oder wenn eine neue Version erscheint – zeigt **Dateien → MicroPython installieren…** eine Anleitung. Kurz gefasst:

1. Die Firmware-Datei (`.uf2`) herunterladen.
2. Den PicoBoy per USB anschließen, die Taste **BOOT** gedrückt halten, kurz **RES** drücken, BOOT loslassen.
3. Am Computer erscheint ein Laufwerk **RPI-RP2**. Die `.uf2`-Datei darauf ziehen.
4. Das Laufwerk verschwindet, der PicoBoy startet neu. In PBC+ Studio **Verbinden** klicken.

> Beim Aufspielen der Firmware wird **alles** auf dem PicoBoy gelöscht, auch deine Programme und Bilder. Sichere vorher mit **Dateien → Flash-Backup erstellen (.uf2)…** den ganzen Speicher als Datei. Das dauert einige Minuten. Dieses Backup spielst du bei Bedarf genauso auf wie die Firmware (Schritt 2 und 3).

